Communicating Process Architectures 2007
Alistair McEwan, Steve Schneider, Wilson Ifill and Peter Welch (Eds.)
IOS Press, 2007

CSP and real-time – reality or an illusion?

Bojan ORLIC, Jan F. BROENINK
Control Engineering,
Faculty of EE-Math-CS, University of Twente
P.O.Box 217, 7500AE Enschede, the Netherlands
{B.Orlic, J.F.Broenink}@utwente.nl

Abstract. This paper deals with the applicability of CSP in general and SystemCSP, as a notation and design methodology based on CSP, in particular in the application area of real-time systems. The paper extends SystemCSP by introducing time-related operators as a way to specify time properties. Since SystemCSP aims to be used in practice of real-time systems development, achieving real-time in practice is also addressed. The mismatch between the classical scheduling theories and CSP paradigm is explored. Some practical ways to deal with this mismatch are presented.

Keywords. SystemCSP, CSP, real-time.

Introduction

Concurrency is one of the most essential properties of reality as we know it. We can perceive that in every complex system, many activities are taking place simultaneously. The main source of complexity in designed systems stems actually from the simultaneous (concurrent) existence of many objects, events and scenarios. Better control over the concurrency structure should therefore automatically reduce the problem of complexity handling. Thus, a structured way to deal with concurrency is needed. CSP theory [1, 2] is a convenient tool to introduce a sound and formally verifiable concurrency structure in designed systems. Our SystemCSP [3] graphical notation and design methodology, as well as its predecessor GML [4], is built on top of the CSP theory. SystemCSP is an attempt to put CSP into practical use for the design and implementation of component-based systems.

Various approaches attempt to introduce ways to specify time properties in CSP theory [1, 2]. SystemCSP as a design methodology based on CSP and intended to be suitable for the real-time systems application area, offers a practical application of those theories. The way in which time properties are introduced in SystemCSP also makes a connection between the two referenced approaches of theoretical CSP.

Specifying time properties is one part of the problem. It allows capturing time requirements and execution times. In practical implementations, resulting time behavior of processes is also the consequence of time-sharing of a processor or network bandwidth between several processes. This time-sharing implies switching the context of execution from one involved process to another, where the order of execution is based on some kind of priority assignment.

Classical scheduling theory offers recipes to give real-time guarantees for systems where several tasks share the same processing or network resource using some priority based scheme. However, as it will be illustrated in Section 2.1.3, there is an essential mismatch between the programming paradigm assumed by classical scheduling techniques and the one offered by the CSP way of design. This mismatch raises the fundamental

question: are CSP-based systems suitable for usage in real-time systems or for this application area should one rely on some other method? This paper will attempt to show possible directions in solving the problem of achieving real-time in CSP-based systems. The first direction in addressing this problem is constructing CSP-based design patterns that can match the form required by the classic scheduling techniques. The second direction is oriented towards the creation of scheduling or real-time analysis theories specific for CSP-based systems.

1. Time properties in specification of CSP-based systems

1.1 Discrete time event ‘ tock’

In [1], time properties are specified by introducing an explicit time event named ‘`tock`’. This implicitly introduces the existence of a discrete clock that advances the time of the system for one step with each occurrence of the `tock` event. Time instants can thus be represented by a stream of natural numbers, where every occurrence of the `tock` event can be considered to increase the current time for one basic time unit. All processes with time constraints synchronize with the progress of time by participating directly in the `tock` event, or via interaction with processes that do. Advantages of this approach are that it is simple, easy to understand and flexible. It does *not* introduce any theoretical extensions to CSP theory and thus formal checking is possible using the same tools (FDR) as in untimed CSP.

1.2 Timed CSP

Timed CSP [2] extends CSP theory by introducing ways to specify time properties in CSP descriptions. There is, however, (yet) no tool that can verify designs based upon Timed CSP.

Times associated with events are non-negative real numbers, thus creating a dense continuous model of time. This assumption makes the verification process complicated and *not* practical. The difference between this approach and introducing the explicit time event (“tock”) is comparable to the difference between continuous systems and their simulation on a computer using discretized time.

The method is also not related to real-time scheduling. It defines the operational semantics for introducing time properties in CSP-based systems. Several essential extensions to CSP are basis for making a system of proofs according to the ones that exist in basic CSP theory. Newly introduced operators include: observing time, evolution transition, timeout operator, timed interrupt operator and time delay.

Time can be observed at any event occurrence. The observed time can then be used in a following part of process description as a free variable.

The expression

$$P = ev1@t1 \rightarrow ev2@t2 \rightarrow display(t2\text{-}t1) \rightarrow SKIP$$

specifies that the time of occurrence of event ev1 is stored in variable t1 and the time of occurrence of ev2 is stored in variable t2. Afterwards a function is called that displays the time interval between the occurrences of event ev1 and event ev2.

The timeout operator is a binary operator representing the time-sensitive version of the *external choice* operator of CSP. It is offering the choice between the process specified as its first operand and the process specified as second operand. In case when a timeout event

takes place before the process guarded via the timeout operator engages in some external event, the control is given to the process specified as second operand.

The expression

$$(ev1 \rightarrow P1) \overset{d}{\triangleright} Q$$

specifies that if the event `ev1` takes place in `d` time units from the moment it is offered, then the process will subsequently behave as process P1. Otherwise, it will behave as process Q.

The *timed interrupt* is a binary operator representing the time-sensitive version of the *interrupt operator* of CSP. The main difference is that the event that triggers the interrupt is actually a timeout event. The process specified as the second operand will be executed after the timeout event signifies that the guarded process did not succeed to successfully finish its execution in the given time interval. As opposed to the timeout operator that uses timeouts to guard only a single event, the timed interrupt operator is guarding the completion of a process. If that process does not finish its execution in the predefined time interval, its further execution is abandoned.

The expression

$$(ev1 \rightarrow P1) \triangle_d Q$$

specifies that the process ev1->P1 will be granted a time interval of d time units to be performed. In the moment when the given time interval expires, further execution of the process ev1->P1 is aborted (interrupted) and the process Q is executed instead.

Introducing *time delay* (*delay event prefix* in Timed CSP) is a step from the world of ideal computing devices capable of infinitely fast parallel execution (as assumed by CSP) to the world of real target implementations. Time delay is used to extend process descriptions with the specification of execution times. In software implementations, the execution times get certain values depending on the processing node which executes a process. During this delay time, a process cannot engage in any event, that is, it acts as a STOP process. In fact, specifying the *delay event prefix* is equivalent to applying the timeout operator on a STOP process as the first operand and the rest of original process as the second operand. A *delay event prefix* is specified by augmenting an event prefix arrow with a time delay value. Instead of single number denoting a fixed execution time, it is possible to specify an interval for the expected time delay. In that case, a pair of values is grouped via square brackets.

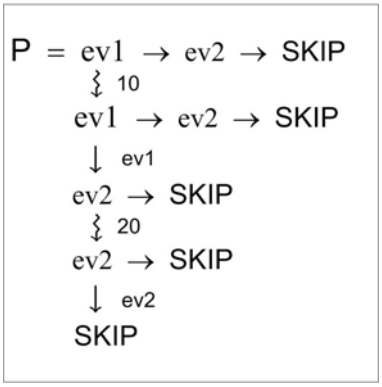
The expression

$$P = ev1 \overset{10}{\rightarrow} ev2 \overset{[10,20]}{\rightarrow} SKIP$$

specifies that after occurrence of event ev1, process P is for 10 time units unable to participate in any event. After the interval of 10 time units expires, process P will offer event ev2 to the environment. Then, after the event ev2 is accepted by the environment, it will take between 10 and 20 time units before process P can successfully finish its execution.

Evolution transition is a way to display an observed delay between events in some particular execution of the process description.

The expression



represents an execution in which the event ev1 has taken place 10 time units after it was initially offered to the environment, and where the event ev2 has taken place 20 time units after it was initially offered to the environment.

1.3 Specification of time properties in SystemCSP

SystemCSP recognizes that operators introduced in TimedCSP are practical for describing time properties of systems. However, we are aware that there is no real need to introduce a dense continuous model of time for modelling software and hardware implementation of processes. Therefore, in SystemCSP we start with the discrete notion of time as in [1] and introduce the basic event `tock` produced by the timing subsystem in regular intervals. Upon the `tock` event, we construct a process that implements a timing subsystem. This subsystem provides services used in the implementation of the higher-level design primitives that provide functionality analogue to the one defined by the timeout and the timed interrupt operators defined in TimedCSP [2]. In this way, it is possible to create designs using Timed CSP-alike operators, to describe them in basic CSP theory, making these designs amenable to formal verification equally as untimed CSP designs.

Section 1.3.1 introduces a notation for specifying time constraints and delays in SystemCSP. Section 1.3.2 provides design patterns for implementation the timing subsystem based on the `tock` event. Sections 1.3.4 and 1.3.5 provide graphical symbols for specification and design patterns for implementation of behaviours defined as in timeout and timed interrupt operators of TimedCSP.

1.3.1 Execution times and time constraints

In the control flow oriented part of SystemCSP, a process description starts with its name label (e.g. name label P in Figure 1). Control flow operators match the CSP set of operators, and are used to relate event-ends and nested process blocks, specifying in that way the concurrency structure of the process. The prefix operator of CSP is represented with an arrow, parallel composition and external choice are represented with a pair of FORK and JOIN elements marked with the type of operator, and so on.

SystemCSP specifies time properties inside square brackets positioned in a separate node element or next to the element they are associated with (see Figure 1). In Figure 1, the first block specifies that the process P is to be triggered in precisely periodic moments of time, with the period equal to Ts. The occurrence of event `ev1` is a point when time is

stored to the variable `t1`. The time when the event `ev2` occurs is stored in the variable `t2`. The keyword `time` is used to denote the current time in the system.

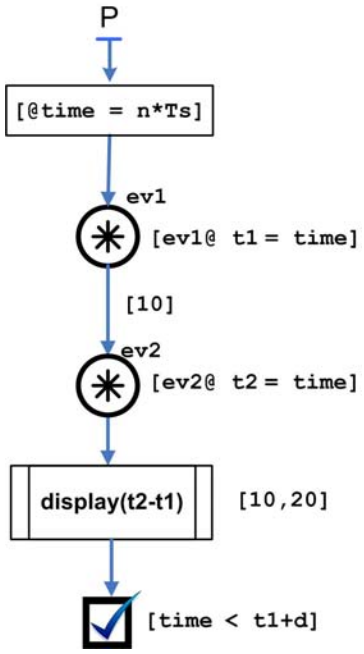


Figure 1 Specifying time requirements

Execution times can also be visualized on SystemCSP diagrams. In SystemCSP, as in Timed CSP, the time delay is specified inside square brackets and instead of a single value representing a fixed execution time, it is possible to display a pair of values that defines a range. The range of possible execution times is bounded by the *minimum execution time* (minET) and the *worst case execution time* (WCET). In addition, often it is useful to keep track of the *average execution time* (avET). In that case, a triple is specified.

The position of the time delay specification is related to the associated diagram element (e.g. next to the associated computation process block or the prefix arrow replacing it or the event that allows progress of the following computation block). The specified delay can be just a number, in which case the default time unit is implied. Otherwise, the specification of time delay should also include a time unit. Time delay can also be specified as a variable that evaluates to some time value.

The *evolution transitions* of Timed CSP are not represented in SystemCSP so far, since they are used for visualizing time delays in observed actual executions of processes. In future, when a prospective tool that can simulate or display execution is built, it can use the same symbol as in Timed CSP.

In addition to operators defined in Timed CSP, SystemCSP also introduces a notation for visual specification of time constraints. Those constraints are not directly translated to the CSP model of the system. These time constraints specify that certain events take place before some deadline or precisely at some time. A deadline can be set relative to some absolute time or as a maximally allowed distance in time between the occurrences of two events. The deadline constraints are independent of the platform on which they are executed. In Figure 1, process P is scheduled to be triggered periodically at precise moments in time. The time constraint associated with the termination of process P specifies that it should take place strictly less then d time units after `t1` moment in time, or, in other

words, process P must finish successfully at most `d` time units after an occurrence of the event `ev1`.

1.3.2 Timing subsystem

Figure 2 introduces one possible design of a timing subsystem. The purpose of this example is *not* to provide a ready-to-use design, but rather to illustrate the point of constructing a timing subsystem starting with the `tock` event. Note that in SystemCSP the symbol * is used to mark the event-end that can initiate an occurrence of the event, while the symbol # is used on the side that can only accept events. From a CSP point of view, this is irrelevant because event occurrence is symmetrical. However, in a design it is often very useful to provide additional insight by specifying the difference between the side that can initiate events and the side that can only accept events.

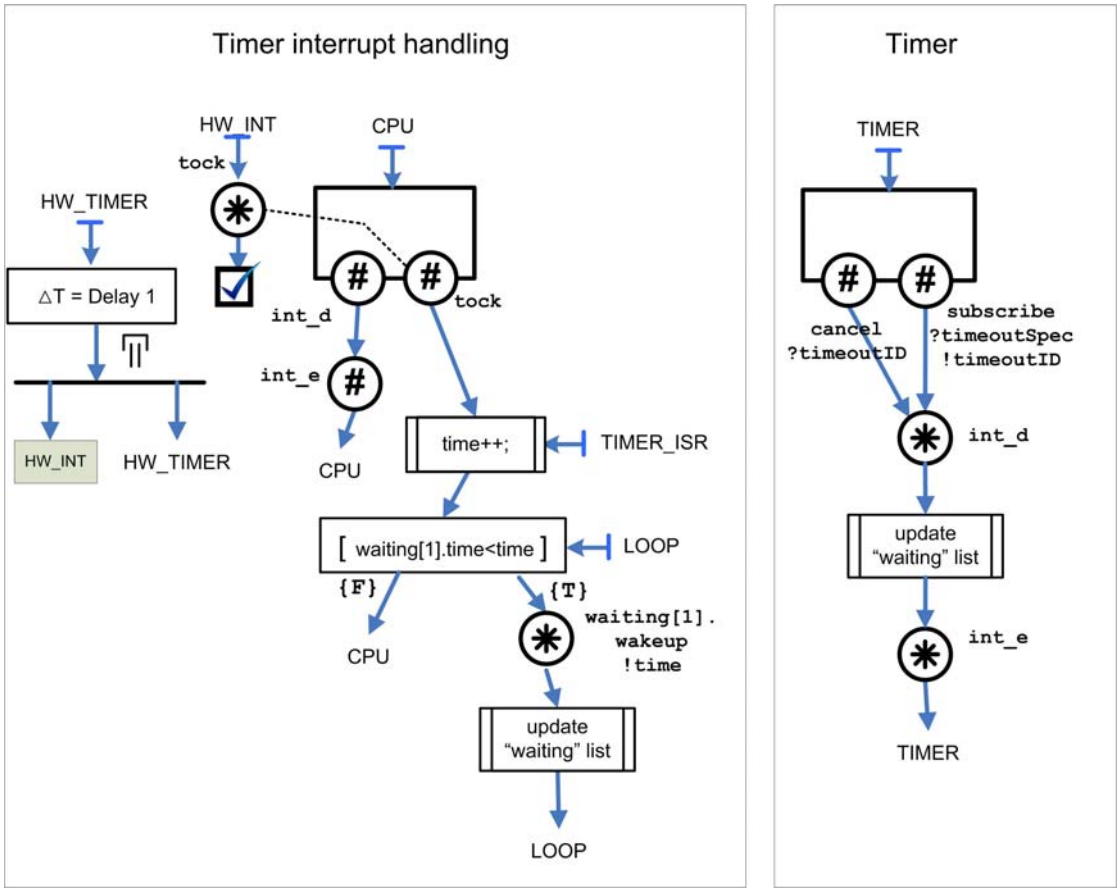


Figure 2 Timing susbsystem

The timing subsystem in Figure 2 contains several processes executed concurrently. HW_TIMER is a process implemented in hardware that forks instances of the hardware interrupt process, `HW_INT`, in regular intervals. The `HW_INT` process synchronizes with the CPU on the event `tock`, invoking in that way the timer interrupt service routine (TIMER_ISR process). Process TIMER_ISR increments the value of the variable `time`. TIMER_ISR also maintains a sorted list of processes waiting on timeout events. Processes in this list, for which the time they wait for is less then or equal to the current time, will be awakened using the `wakeup` event. The awoken processes will be removed from the top of the list. In the case the awoken process is periodic, it is added again to a proper place in the

waiting list.

The process CPU acts as a gate that can disable (event `int_d`) or enable (event `int_e`) the timer and other interrupts. When interrupts are enabled, event `tock` can take place and as a consequence the interrupt service routine TIMER_ISR will be invoked. The case of occurrence of the `int_d` event, as a next event only the `int_e` event is allowed and until then, the event `tock` cannot be accepted, and consequently interrupts are *not* allowed to happen.

Processes using services of the timing subsystem can, via the TIMER process, either subscribe (via event `subscribe`) to the timeout service or generate a `cancel` event to cancel a previously requested timeout service. Since these activities are actually updating the waiting list, this list must be protected from being updated in the same time by TIMER and TIMER_ISR processes. That is achieved in this case via disabling/enabling interrupts (`int_d` / `int_e` events).

1.3.3 Watchdog Design Pattern

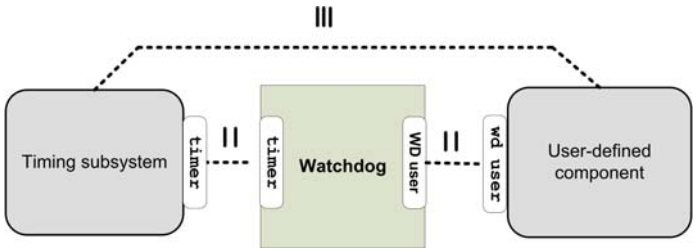


Figure 3 Interaction diagram: using a watchdog interaction contract

The interaction view specified in Figure 3 illustrates the interaction between a user-defined component and the timing subsystem component via the watchdog interaction contract. The watchdog pattern is used to detect timing faults and to initiate recovery mechanisms.

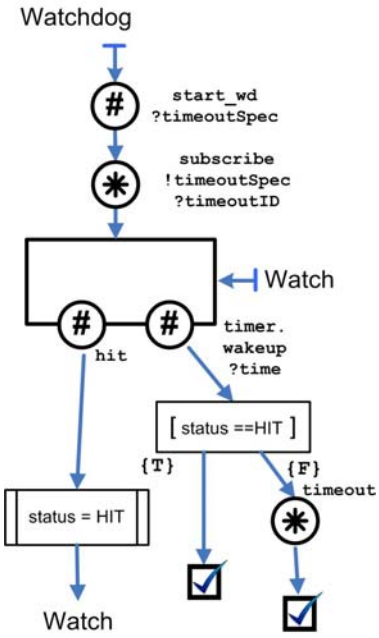


Figure 4 Watchdog design pattern

The design pattern for the watchdog process (see Figure 4) relies on services provided by the timing subsystem. A user initializes the watchdog using the `start_wd` event, which results in a watchdog request to be notified by the timing subsystem when the specified timeout expires. In case when the watchdog user chooses to initiate the `hit` event, the watchdog is disarmed. Otherwise, upon occurrence of the timeout (event `wakeup`), the watchdog will initiate the `timeout` event, invoking the warning situation.

1.3.4 Timed interrupt operator

The timed interrupt operator is simply a time-sensitive version of the interrupt operator. Its implementation, as depicted in Figure 5, contains the interrupt operator, and additional synchronization with a watchdog process. The watchdog is initialized, via the `start_wd` event, with the timeout value specified in the timed interrupt operator. When the guarded process (`ev1->P` in the example of Figure 5) finishes and the hit event takes place, the associated watchdog process will be disarmed. If, however, the timeout event takes place, it will cause the guarded process to be aborted, and the process specified as the second operand (process Q in the example of Figure 5) is executed.

The closed dotted line at the left-hand side of the Figure 5 encircles elements that are providing the implementation of the behaviour specified by the timed interrupt operator. The right-hand side of the Figure 5 abstracts away from those implementation details by providing a way to specify the timed interrupt operator as basic element of the SystemCSP vocabulary. In fact, a pair of blocks with timed-interrupt symbol is used to determine the scope of the operator, similarly as brackets are used in CSP expressions.

$$(ev1 \rightarrow P1) \triangle_d Q$$

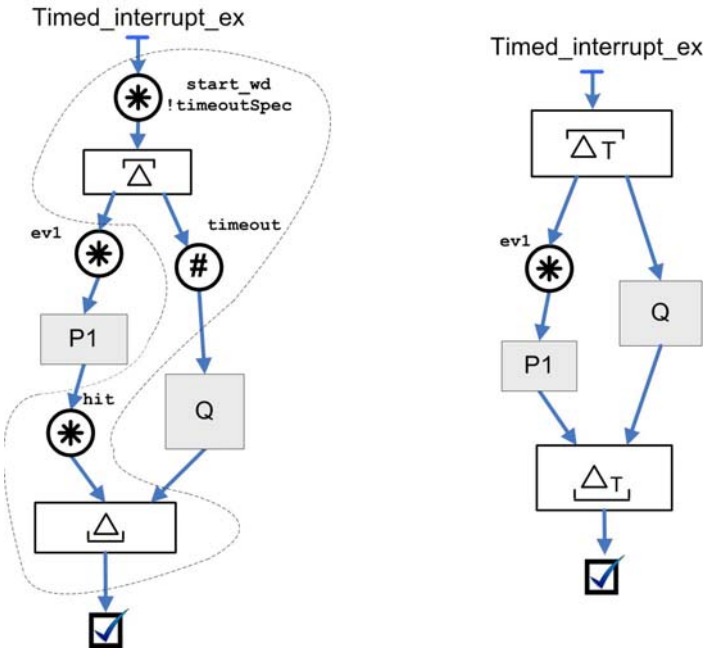


Figure 5 Timed interrupt – implementation and symbol

1.3.5 Implementation of the Timeout operator

The timeout operator is simply a time sensitive external choice where one of the branches is a guarded process and the other one starts with a time event that will be initiated by the associated watchdog after the requested timeout expires. Following the timeout event (see Figure 6), the process specified as second operator is executed.

$$P = (ev1 \rightarrow P1) \overset{d}{\triangleright} Q$$

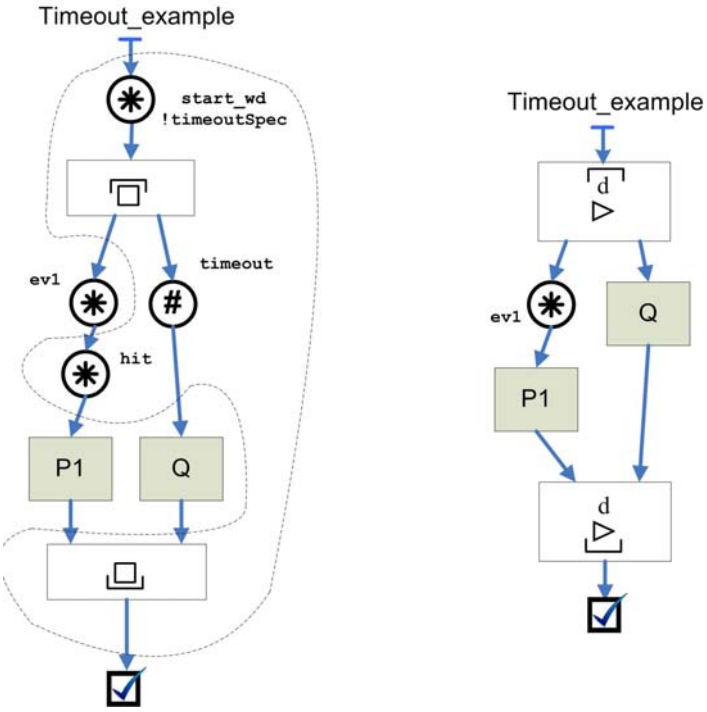


Figure 6 Timeout operator – implementation and symbol

Figure 6 depicts the implementation and a symbol of the timeout operator on a simple example and its associated visualization using the symbol for timeout operator. Instead of the letter d inside the timeout operator symbol, it is possible to use any number or variable representing time. The left hand-side of Figure 6 depicts the implementation details encircled via the dotted line, while the right-hand side introduces notation elements used to represent the timeout operator as one of the basic building blocks in SystemCSP.

2. Real-time in the implementation of CSP-based systems

2.1 Identifying problems

2.1.1 Origin of time constraints in implementation of control systems

An *embedded control system* interacts with its environment via various sensors and actuators. Sensors convert analogue physical signals to signals understandable by the embedded control system (digital quantities in case of computer-based control). Actuators (motors, valves….) perform transformation in opposite direction (note in Figure 7 the different types of arrow symbols used for digital and analogue signals).

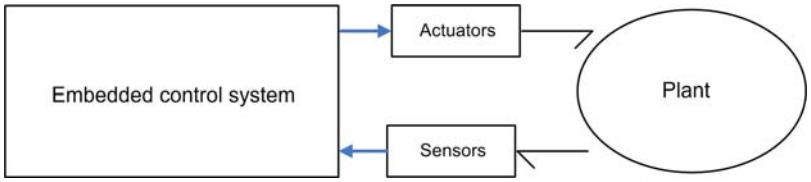


Figure 7 Typical control system

Obviously, a (control) system and its relevant environment (plant, i.e. machine to be controlled) exist concurrently. In fact, both plant and control system are often decomposed in subsystems that exist concurrently and are cooperating to achieve the desired behaviour. Thus, in the control system application area, concurrency is naturally present.

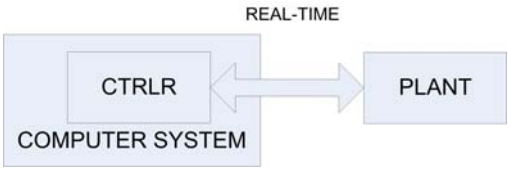


Figure 8 Implementation of computer control system

Figure 8 illustrates a computer implementation of a control system. The control algorithm (CTRLR block in Figure 8) is performed by the computer system. In general case the computer system can contain many computer nodes connected via some network. The time pattern of the interaction between the control system and its environment is based on the time constraints imposed by the underlying control theory. The computer system implementing the embedded control system must be able to guarantee that the required time properties will be met in real time. So, in real-time systems, “correctness of the system depends not only on the logical result of the computation but also on the time at which results are produced” [5]. In those systems, the response should take place in a certain time window. Real-time is not about the speed of the system, but rather about its relative speed compared to the required speed of its interaction with the environment. Rather than fast, the response of those systems should be predictable. A fast system will not be real-time if its interaction with environment requires a faster response. A slow system can work in real-time if it is faster than its interaction with environment requires.

A control loop starts with sensor data measurements and finishes with delivering command data to the actuators. The time between two subsequent measurement (sampling) points is named *sampling period* and the time between a sampling point and the related actuation action is named *control delay* [6]. Digital control theory assumes equidistant

sampling and fixed control delay time. On an ideal computer system, the control loop computation is performed infinitely fast. In reality, it takes a certain time that should be bounded. This gap between ideal and real computing devices reflects itself in a design choice between two possible patterns used in practice for ordering sampling and actuation tasks.

In the Sample-Compute-Actuate approach, depicted in upper part of Figure 9, the computation time is usually assumed to be negligible, implying the computing device close to ideal. Rule of thumb is that the behaviour of a control system will still be acceptable when this computation time is kept smaller then around 20% of the sampling period. Obviously this approach does not really guarantee that system will always work as expected by control engineers. Especially in complex control systems that contain more than one control loop, or control loops closed over a network, the influence of variable control delay becomes an important factor in the resulting behaviour of the control system.

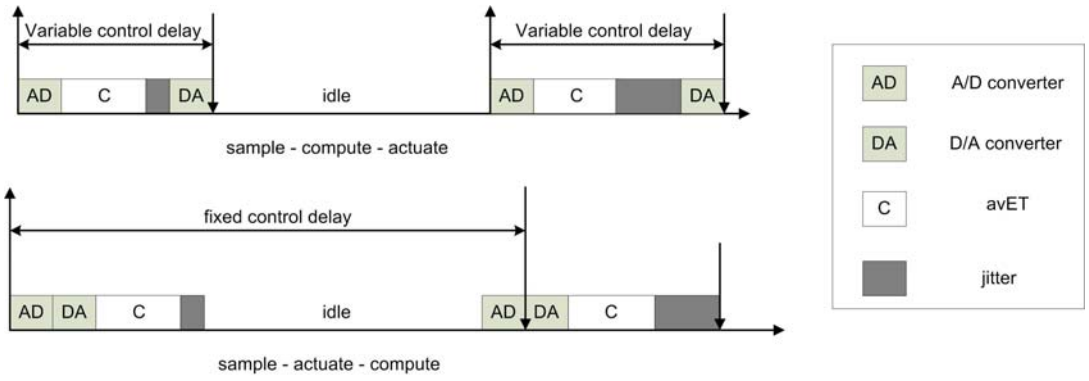


Figure 9 Sampling period and control delay (adapted from [7])

The second approach, Sample-Actuate-Compute, takes into account the non-ideal nature of the computation devices. In the approach depicted in the lower part of Figure 9, the control delay is fixed and usually set to be equal to the period. By fixing the point of actuation to be immediately after the sampling point for the next iteration, two goals are achieved: first, actuators are prevented from disturbing the next cycle of the input sampling and second, the control delay is fixed, which allows compensating for it in the control algorithm using standard digital control theory.

From these temporal requirements imposed on control theory, real-time constraints are imposed on the implementation of control systems. In both described approaches, a constant sampling frequency is achieved by performing the sampling tasks in *precisely periodic* points of time. In the first approach, the computation and actuation tasks need to get processor time as soon as possible, resulting in assigning them high priority value. The relative deadline of this task can be set using the aforementioned rule of thumb, to be 20% of the sampling period. In the second approach, as a consequence of fixing the actuation point in time, a hard real-time deadline is introduced for the computation task.

2.1.2 Classical scheduling theories

Real-time scheduling is nowadays a well-developed branch of computer science. It relies on the programming model where tasks communicate via shared data objects protected from the simultaneous access via a locking mechanism. Good overview of the most commonly used scheduling methods is given in [5].

Time constraints are in real-time systems met by assigning different priority levels to

the involved tasks according to some scheduling policy. E.g. in Earliest Deadline First (EDF) scheduling, the task with a more stringent time requirement will get a higher priority. In Rate Monotonic (RM) scheduling, a process with higher sampling frequencies will have higher priority. Good comparison of all advantages and disadvantages of EDF and RM is given in [8]

An emerging way to schedule tasks in control systems is presented in [9]. In there, it is demonstrated that, compared to EDF and RM, better performance of a control system can be achieved when priorities are dynamically assigned according to the values of some control-system performance parameters.

Designs based on shared data objects as assumed in classical scheduling theories differ from designs based on buffered communication, or rendezvous-based communication, as assumed in CSP. In communication via shared data objects, *no* precedence constraints (set of "before"/"after" relationships between processes specifying set of relative orderings of the involved tasks) are introduced by the communication primitives.

2.1.3 Fundamental mismatch between CSP and classical scheduling – Communication induced precedence constraints

A rendezvous synchronization point introduces a pair of precedence constraint dependencies. In Figure 10, the control flow specifies that process A must be executed before process C and process B before process D. In addition, due to rendezvous synchronization on event ev1, subprocess A must be executed before subprocess D and subprocess B must be executed before subprocess D. In right-hand side part of the Figure, this is illustrated by dashed directed lines specifying precedence constraints from subprocess A to subprocess D (let us abbreviate this with A->D) and from subprocess B to subprocess C (B->C). Note that A, B, C and D are processes that can contain events and synchronize with the environment.

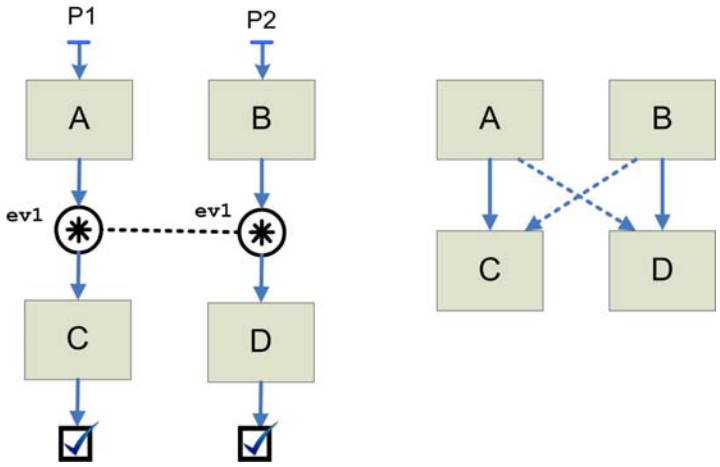


Figure 10 Rendezvous communication introduces new precedence constraints

In Figure 11, the communication from process P1 to P2 is buffered via an intermediate buffer process. Precedence relations are visualized as oriented dashed lines.

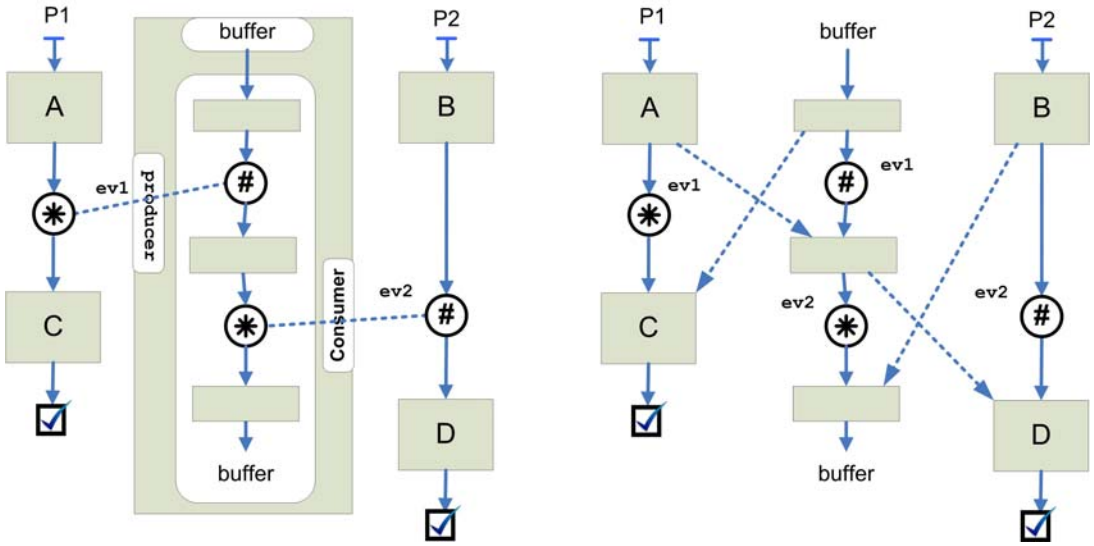


Figure 11 One-place buffer

As it can be seen in Figure 12, only the precedence dependency from A to D (A->D) exists, since the data must be present before it can be consumed. If the data flow direction for buffered asynchronous communication was from P2 to P1 then only the B->C precedence constraint would exist.

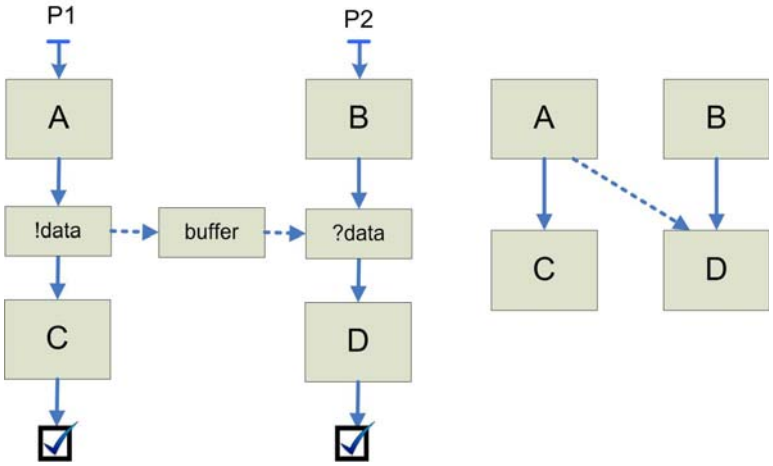


Figure 12 Precedence constraints for buffered communication

If, however, communication is via shared data objects (see Figure 13 and Figure 14), *no* precedence constraints are involved. The reason is that the shared data object has the semantics of an overwrite buffer, where a consumer always consumes the last fresh value available. In fact, in this case, it is more appropriate to use the term reader instead of the term consumer.

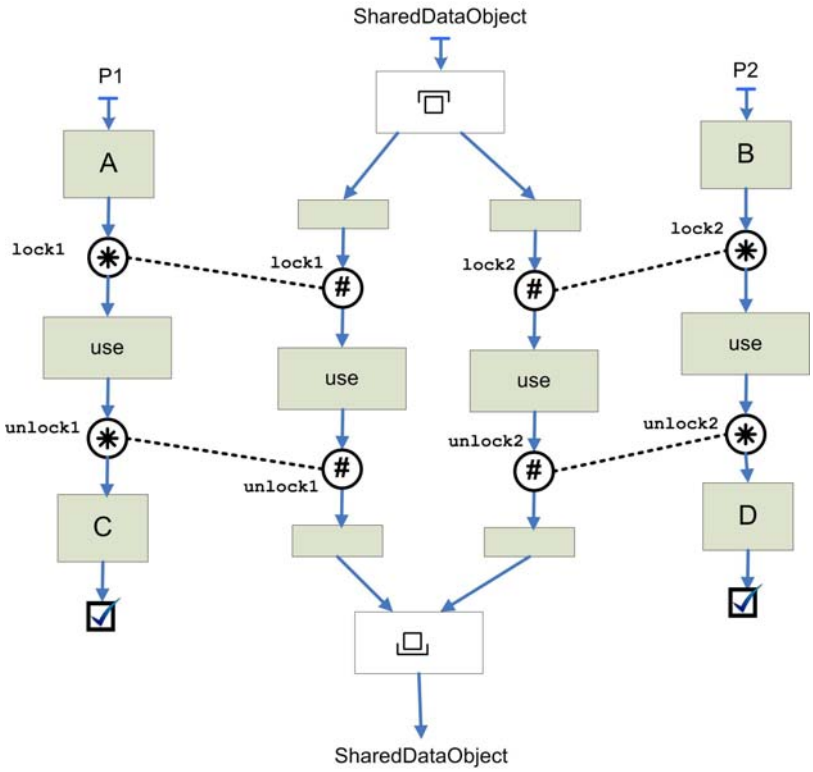


Figure 13 Shared data objects

Note that in case of shared data object communication, a process can still be blocked on waiting to access the shared data object. Scheduling theories do take into account this delay by calculating the worst-case blocking time.

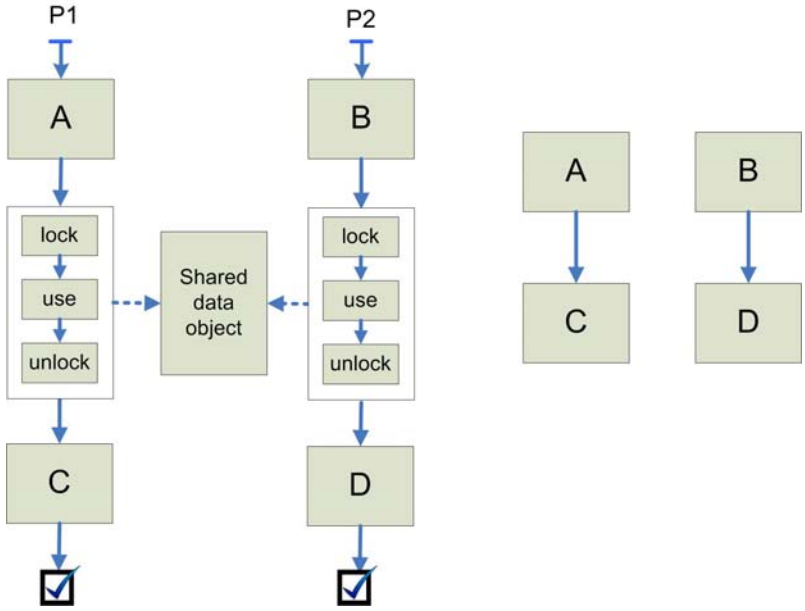


Figure 14 Precedence constraints in shared data object communication

In Figure 10, usage of rendezvous channel yields the possible orderings of subprocesses: (A || B) ->(C || D). Symbol A || B is used to abbreviate that A and B can be executed in any order, which is equivalent to composing them in parallel. When an

asynchronous channel is used, it is equivalent to erasing one of the two precedence constraints (depending on the direction of data flow as explained above) and the resulting set of possible orderings is larger, allowing e.g. A->C->B->D (after A process P1 writes to the buffer and continues), which is not covered originally. When shared data objects are used, the set of possible orderings is even larger because another precedence constraint is removed. Thus, relaxation of precedence constraints, introduced by changing the type of applied communication primitive, leads to extending the set of possible behaviours.

2.1.4 Influence of assigning priorities on analysis

In systems with rendezvous-based communication, using priorities reduces the set of possible traces only in pathological cases [10]. For instance, consider the system given in Figure 15, with the assumption that the highest priority level is assigned to process P1, the middle one is assigned to P2 and the lowest priority level is assigned to process P3. This priority ordering can be for instance implemented using a PriParallel construct, or alternatively, the system can rely on absolute priority settings. In any case, the relative priority ordering is from P1 to P3.

Priorities defined in this way will tend to give preference to P1i blocks compared to P2j blocks and also will give preference to P2j blocks compared to P3k blocks, but in fact the real order of execution can be any depending on the order of events accepted by the environment. Thus, the set of possible orderings of processes P1i, P2j, P3k and the set of related event traces is in general case not reduced by a priority assignment.

A PriAlternative construct is in fact giving relative priorities to event ends participating in the same PriAlternative construct. Those priorities are used only in the case when more then one event is ready. Again, the environment determines what events will be ready in run-time and thus as for PriParallel all traces are still possible.

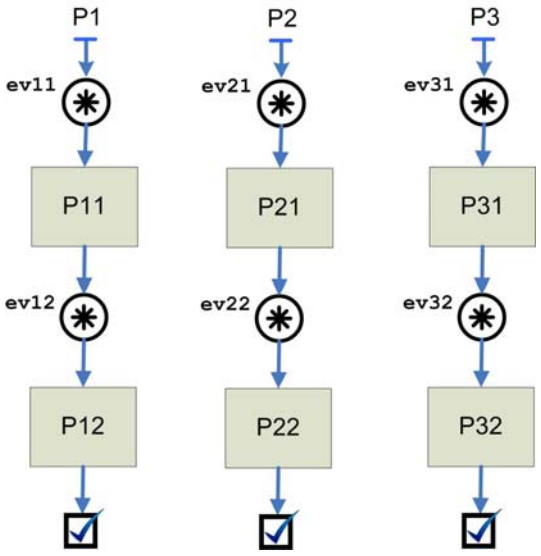


Figure 15 Some processes executed in parallel

From the discussion above, it is clear that assigning priorities to processes does not reduce the set of possible traces. Thus, to guarantee real-time, it should be verified that constraints are satisfied along any possible trace in the system. One reasonable approach for checking real-time guarantees is systematically replacing every Parallel composition with an equivalent automaton and associating execution times and deadlines with points in the

control flow of that equivalent automaton. This approach is subject of discussion in Section 2.3.2.

The conclusion is that structuring a program in the CSP way does influence schedulability analysis, because rendezvous-based communication makes processes more tightly coupled due to the additional precedence constraints stemming from the rendezvous synchronization. In rendezvous-based systems, priority of a process does not influence dominantly the order of execution. The actual execution ordering in the overall system is dominantly determined by the communication pattern encapsulated in the event-based interaction of processes. This interaction pattern inherent to the structure of the overall system, will due to the tight coupling on event synchronization points, always overrule the priorities of involved processes. Assigning a higher priority to one process, engaged in a complex interaction scheme with processes of different priorities, does not necessarily mean it will be always executed before lower-priority processes. This situation can also be seen as analogue to the priority inversion phenomenon in classical scheduling theory.

2.1.5 Priority inversion and using a buffer process to alleviate the problem

In classic scheduling, priority inversion is a situation where a higher-priority task is blocked on a resource held by a lower-priority task, which has as a consequence that tasks of intermediate priority are in the position to preempt the execution of the lower priority task and in that way prolong blocking time of the higher priority process. Let us try to view rendezvous channels in CSP-based systems as analogue to the resources shared between tasks in classic scheduling. In that light, waiting on a peer process to access the channel can be seen as analogue to the blocking time spent waiting on a peer task to free the shared resource. Viewed in this way, the system consisting of processes P1, P2 and P3 composed via a PriParallel construct depicted in Figure 16 illustrates the priority inversion problem. Process P1 has to wait for process P3 to enable occurrence of event ev3 and in meantime process P2 can preempt P3 and execute although its priority is lower than the one of P1. In this figure, the control flow of each process is given by concatenating event-ends and subrprocesses. The arrows with numbers on the left side of every process, indicate the actual order of execution that takes places due to the interaction pattern and despite the specified set of priorities.

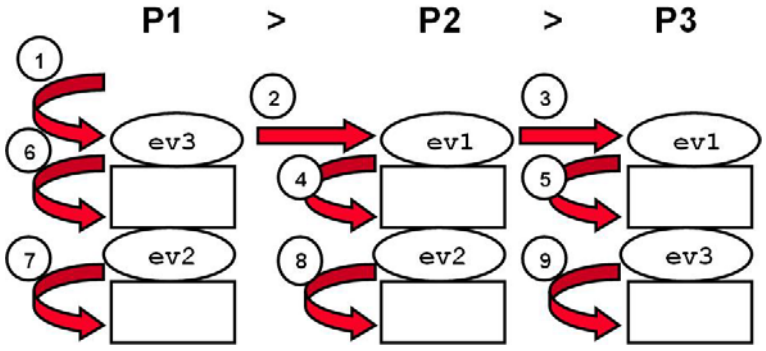


Figure 16 Process execution is dominantly determined by communication patterns rather then by process priority

Buffered channels are proposed in [4] to alleviate this problem. There, a proof of concept is given by implementing the buffered channel as a CSP process. However, a buffer process does not help when the process of higher priority (P1 in the example) is playing the role of a consumer and the process of lower priority (P3) is playing the role of a producer. In that case, the direction of the data flow introduces the precedence constraint

from the event end in process P3 to the event end in process P1. The priority of the buffer process is assumed to be equal to the priority of the higher-level process for this scheme to work. Priority inversion in rendezvous-based systems is caused by precedence constraints leading from a process of lower priority to a process of higher priority. Using high-priority shared-data object communication primitives between processes of different priorities eliminates the priority inversion problem.

2.1.6 Absolute versus relative specification of priorities

Classic operating systems offer usually a fixed range of absolute priority values that can be assigned to any of the tasks/processes. In occam and the CT library, the concept of PriParallel construct is introduced that allows one to specify relative priorities instead of absolute ones. The index of a process inside a PriParallel construct determines its relative priority compared to the other subprocesses of the same construct. A program shaped as a hierarchy of nested PriParallel and Parallel constructs, results in an infinite number of possible priority levels. This approach also offers more flexibility, since new components can be added on proper places in the priority structure without the need to change priorities of already existing components.

However, while absolute priority ordering guarantees that any two processes are comparable, this is not the case in Par/Pripar hierarchies. Let's consider following example:

```
PAR
  PRIPAR
    A
    B
  PRIPAR
    C
    D
```

Where as the two PriPars define A as of having higher priority then B and C having higher priority then D, no preference is given to any when for instance B is compared to C, or A to D. They are considered to be of equal priority. So:

```
priority(C) = priority(A) > priority(B) = priority(C)
priority(C) = priority(A) > priority(B) = priority(D)
priority(D) = priority(A) > priority(B) = priority(C)
priority(B) = priority(C) > priority(D) = priority(A)
```

This looks confusing and inconsistent. If only prioritized versions of the Parallel construct (PriPar constructs) were used, there is *no* confusion. In fact, it is like collapsing a big sorted queue into smaller subqueues that can further be decomposed in subsubqueues. Only with a Par being a parent of PriPar constructs, the priority ordering problems appear.

The question is whether the relative priority ordering schemes, as the ones of occam-like hierarchies of Par and PriPar constructs, can be efficiently used in combination with the classical scheduling methods, for instance RM and EDF.

To be able to apply any priority-based scheduling method in a way that will avoid introducing priority inversion problems, as concluded in the previous section, processes of different priorities should be decoupled via consistent usage of shared data objects.

First let us consider the suitability of a PriPar construct for RM scheduling. The problem with RM scheduling is that it is not compositional. This means that if one

composes two components with inner RM based schedulers, the resulting component does not preserve real-time guarantees. Thus, it would not be possible to define a Par of components on top level and to have PriPar based RM schedulers inside each of them. If, however, a hierarchy consisting only of PriPar constructs is used to implement RM priority assignment, this hierarchy can be seen as dividing one big queue into a hierarchical system of subqueues, where strict ordering is preserved. Theoretically for large queues, a hierarchical organization can significantly increase the speed of searching. In practice, however, systems usually do no need more then 8 or 16 or 32 different priority levels, which allow efficient implementation based on a single status register of size 8, 16 or 32 bits and dedicated FIFO queues for every priority level.

The conclusion is that in principle, by using only Prioritized versions of Par constructs and decoupling components with different priorities via shared data objects, it is possible to apply the RM scheme in occam-like systems. Now let us see if occam-like relative priority orderings are suitable for EDF schedulers.

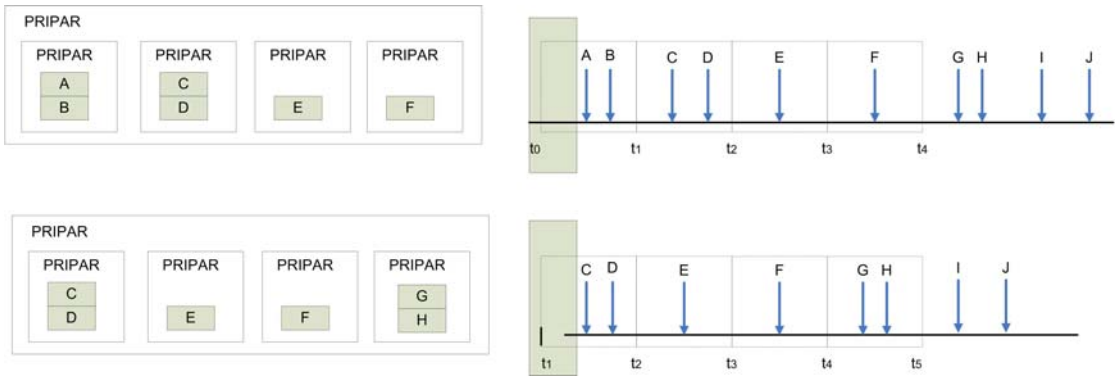


Figure 17 EDF scheduler

Implementing an EDF scheduler is tricky with fixed global priorities because the actual importance of a task is proportional to the nearness of a deadline and this nearness is a factor that keeps changing in time. Trying to assign a global priority to a process whose importance in fact changes with time is unnatural. The occam-like scheduling based on relative priorities could come as more natural solution. For instance one can divide a certain next part of the time axis into several time windows (see the right-hand side of Figure 17). Each time such a window can be associated with a single PriPar construct (compare the left-hand side and the right-hand side of Figure 17). The top level PriPar construct is used to sort the nested PriPar constructs, associated with time windows, according to their time order. Tasks with time constraints are then inserted in the PriPar construct related to the time window where their deadlines fall in. E.g. tasks C and D have deadlines falling into the interval (t1, t2) and are thus in upper part of Figure 17 mapped to the second PriPar construct.

The processes far away in the future and out of scope of any time window will be kept in a separate queue. After all tasks associated with the first time window are processed, this PriPar construct is removed from the top-level PriPar construct. The removed PriPar construct is then reused–it is associated with the first previously not mapped time interval. The non-allocated processes from the far-future queue that fall into that time window (G and H in Figure 17) are now mapped to it and the associated PriPar is now added to the top-level PriPar as the least urgent time window (lower part in Figure 17).

For this scheme to work, again, there should be no precedence constraints among tasks and thus rendezvous or buffered communication is not allowed or it should be taken into

account by deriving intermediate deadlines as in EDF* (see Section 2.2.1).

The described approach for implementing EDF scheduler is based on relative priority orderings. However relative priority ordering is used for a generic implementation of the scheduler and not as a way to specify priorities of user defined processes in the application, as it was the case in occam. In order to apply this scheme in practice, the application itself should specify deadlines and not PriPar constructs or priorities.

A major problem with using relative priorities based on PriPar/Par constructs, is that it hard-codes priorities in the design, while a design of an application should be independent of priority specification. Reason is that a priority level is related to both the time requirements as specified in an application and the time properties of the underlying execution engine framework A choice is to design applications without introducing priorities and to postpone the process of assigning absolute priorities or deadlines (for EDF based scheduling) to the stage of allocation, where it is suited more naturally. Thus, the recommendation is *not* to use PriPar constructs. PriAlternative on the other hand makes sense independently of the scheduling method used.

2.2 Solution direction 1 – classic scheduling

Most straightforward approach to making CSP designs with real-time guarantees is using CSP-based design patterns that match the programming paradigm of classical scheduling.

*2.2.1 EDF**

In classical scheduling techniques, precedence constraints can be specified between tasks and special extensions exist for some scheduling theories (e.g. *modified EDF* - EDF*) to enable them to deal with the precedence constraints. EDF* takes precedence constraints into account by deriving the deadline of a task from the WCETs and deadlines of the following tasks.

When rendezvous channels are seen in the light of the introduced precedence constraints, the EDF* scheduling algorithm is applicable to rendezvous based systems. In applying EDF* to rendezvous based systems, calculation blocks can be considered to be schedulable units. A deadline of a calculation block is updated to the minimum value calculated upwards of any trace starting with some fundamental deadline and leading via the chain of precedence constraints to the current calculation block. The value is calculated starting with the time of the fundamental deadline and substracting WCET of every code block passed while going upwards the trace towards the block whose deadline is being derived in this way.

2.2.2 Design with rendezvous channel communication, convert them to shared data objects when necessary

Regarding solving the priority inversion problem of rendezvous-based system by relaxing the used type of communication primitive, Hilderink [7] states that a deadlock-free program with rendezvous channels will still be deadlock-free when rendezvous channels are substituted with buffered channels. This is in fact intuitively explainable if we realize that deadlock is in fact a circle of precedence constraints (some being due to event prefix and sequential composition operators and some due to rendezvous communication). Since substituting rendezvous channels with buffered or shared data objects removes some of precedence constraints, this can remove some deadlock problems, but cannot introduce new ones.

Thus, a convenient design method could start with a design based on rendezvous

channels. Such initial design is amenable to deadlock checking. After allocation and priority assignment, all rendezvous channels between processes of different priorities can be replaced with shared data objects allowing the usage of classic scheduling techniques, while preserving the results of deadlock checking. However, this approach might not always be feasible. Note that relaxing the precedence constraints associated with rendezvous channels results in an extended set of behaviours by including the possible behaviours that were not formally checked. As a consequence, a new implementation that was produced in this way might not anymore be a refinement of the initial specification. For conformance of such an implementation to its specification both its traces and failures must be subsets of the traces and failures defined in the specification.

2.2.3 Design pattern for implementation of a typical control system

Control systems typically function at a number of levels (see Figure 18). At the lowest level, the highest priority layer is situated. *Safety control* makes sure that functioning of the system will not endanger itself or its environment. It is especially important when embedded control systems are employed in *safety-critical* systems. *Loop control* is a hard real-time part that periodically reads inputs from sensors, calculates control signals according to the chosen control algorithms and uses the obtained values to steer the plant via actuators. *Sequence control* defines the synchronization between the involved subsystems or devices. *Supervisory control* ensures that the overall aim is achieved by using monitoring functions, safety, fault tolerance and algorithms for parameter optimization. *User interface* is an optional layer that supports the interaction of the system with an operator (user), in the form of displaying important part of the system's state to the operator and receiving the commands from the operator.

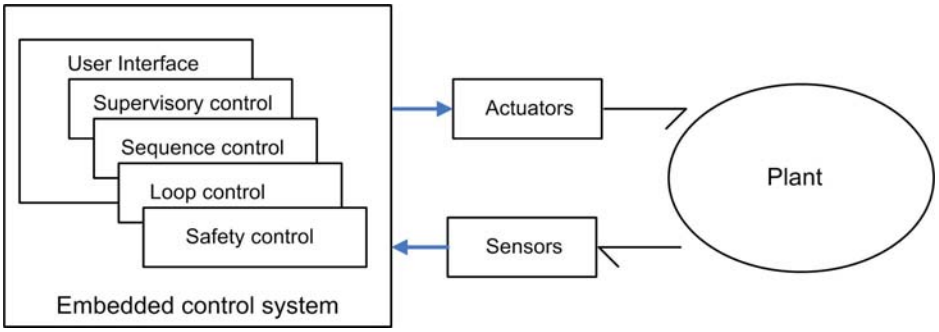


Figure 18 Typical control system

In fact, a complex control system (e.g. a production cell) typically contains several devices that need to cooperate. Every device can participate in any or all mentioned layers. The supervisory and sequence control layers are often event based and the control loop is always time-triggered and periodic, with the period in general different from one device to another. Software components in charge of devices are either situated on the same node or distributed over several nodes.

Figure 19 illustrates data/event dependencies between layers situated in the same device as well as between layers distributed over several or all participating devices. Two ways of clustering subcomponents are possible: horizontal – where a centralized supervisory layer, sequence layer, control loop layer and safety control layer exist, or vertical where parts belonging to the same device are considered to be a single component.

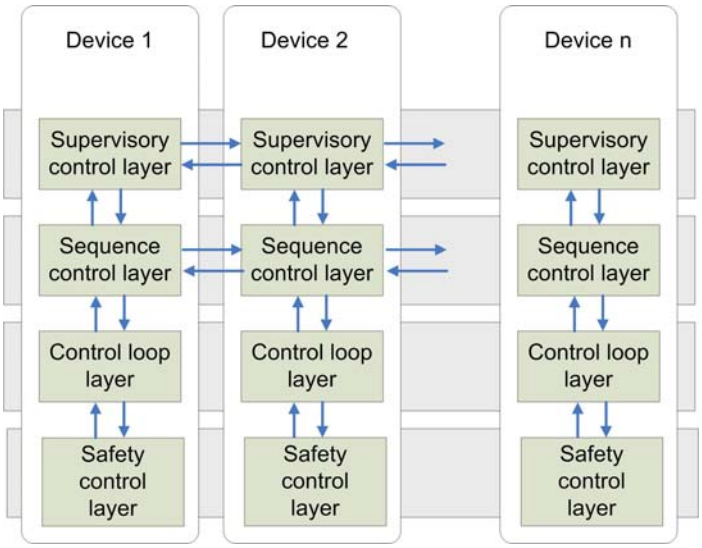


Figure 19 Typical layered structure of complex control system

In SystemCSP, design patterns can be made where either vertical or horizontal groups are structured as components and the orthogonal groupings form interaction contracts. Let us consider the case where devices are treated as components and layers as interaction contracts. Every device is in this approach a component that provides ports, which can be plugged into one of the four interaction contracts: supervision, sequence control, safety, loop control (see Figure 20).

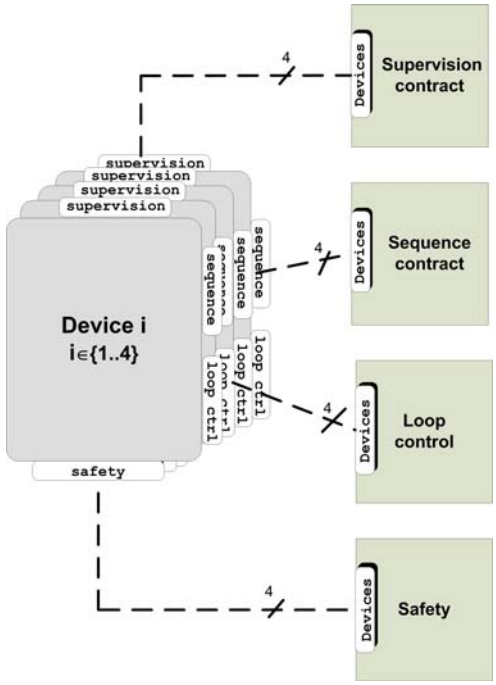


Figure 20 One SystemCSP design pattern for complex control systems

Every contract contains logic for handling several devices and managing synchronization between them. Often, it is useful to merge some of those interaction contracts into a single interaction contract (e.g. some safety measures can be in the

sequence control contract or sequence and supervision layer can be merged).

A loop control contract is often implicit since there is no other dependency between control loops except for common usage of the timing and I/O subsystems to ensure precise in time execution of time-triggered periodic sampling/actuation actions. Upon performing the time triggered sampling/actuation (I/O subsystem) actions, loop-control processes of related devices are released to perform computation of control algorithms. A loop control interaction contract can for instance perform scheduling (RM, EDF) of the involved loop control processes, check whether deadlines are missed and raise alarms to the safety or supervision interaction contract when that happens. E.g. in the overload conditions, a loop control interaction contract can have a centralized policy to decrease the needed total computation time in a way that reduces performance but does not jeopardize the stability of the control system.

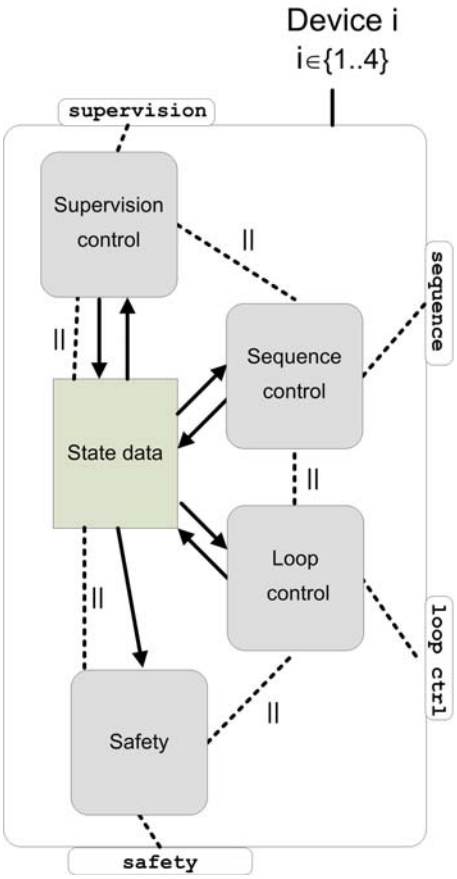


Figure 21 Device internals organization

Internally, a device component might be organized as in Figure 21, with a subcomponent dedicated to the implementation of every role that maps to one of the layers supported by the device, and a subprocess dedicated to maintaining the state data of a device. In Figure 21, in order to put emphasis on structure and data-flow and not on control flow, the GML-like interaction oriented SystemCSP diagram is used, where communication data flows and binary compositional relationships are specified. A centralized process is introduced to manage access to the data that captures the *state* variables of a component. This process is in fact the shared-data object communication pattern that allows decoupled communication between processes implementing roles of various layers. In practice, for efficiency reasons, the state data process can also be implemented as a passive object that provides the necessary synchronization. It can be, for instance, a lock-free double buffered

channel. A centralized process for device's state data access is also convenient in case when the loop control process is replicated for fault tolerance reasons.

In this typical structure, subcomponents get different priorities. The safety layer is of highest priority and is activated only when it is necessary to handle alarm situations. The next range of priorities is associated with loop control subcomponents. Range of priority levels might be necessary for the implementation of the scheduling method that will guarantee their execution in real-time. Sequence control is an event based layer and thus of less importance than the time-constrained loop control layer. The Supervision layer is performing optimization and is thus of least importance.

2.3 Solution direction 2 – developing scheduling theory specific for rendezvous-based systems

If however, the intention is to use rendezvous-based channels/events as basic primitives then a distinct scheduling theory must be developed. The topic of achieving real-time guarantees in systems with rendezvous synchronized communication is a research field that still waits for a good underlying theory.

2.3.1 Event-based scheduling

Figure 16 illustrates that attaching priorities to processes that communicate via rendezvous channels influences the behaviour of the rendezvous-based systems much less then expected.

Instead of trying to apply classic scheduling methods, it is possible to admit the crucial role of events in CSP based systems and assign priorities to events instead of to processes. Deadlines can be seen as time requirements imposed on events or on distances between some events. Furthermore, while priority of processes is local to a node, the priority of an event is still valid throughout the whole distributed system. Such *event-based scheduling* seems to promise a way to get better insight and more control over the way the synchronization pattern influences the execution and overrules the preferred priorities.

Section 2.1.5 has introduced an analogy between the priority inversion problem in classical scheduling methods and the analogue problem in the rendezvous based systems. In classical scheduling, the standard solution to the priority inheritance problem is that the lower-priority task holding the resource needed by the higher-priority task gets a temporary priority boost until it frees the resource. If we apply this analogy to a rendezvous channel as a shared resource, then the peer process is holding the resource as long as it is not ready to engage in rendezvous. Thus to avoid priority inversions, the complete control flow of the lower-priority task, that is taking place *before* the event access point, should get a priority boost. The "before" relation is formally expressed via precedence constraint arrows. Thus, starting from some event end, the priority of all events ends upwards the precedence constraint arrows should be updated to be of equal or higher value. Keep in mind that as explained in Section 2.1.3, extra precedence constraints are introduced on every rendezvous synchronization point. Thus, the process of updating priorities propagates through rendezvous communication points to other processes. Eventually, a stable set of priorities is reached. This set of priorities is in the general case different from the initially specified set of priorities.

The user can set an initial set of preferred priorities to some subset, or to all event ends in the program. For instance, one can initially assign priorities to event ends by assigning priorities to processes, which can for instance result in automatically associating the specified process-level priority with every event end in the process. Those preferred values are used as initial values in the aforementioned procedure of systematically updating

priorities of event ends. The set of priorities obtained by applying this algorithm reveal a realistic or an achievable set of values after the synchronization pattern is taken into account. In this process, priority inversions are inherently eliminated.

In the example of Figure 16, in event-based scheduling, event ends initially get priorities according to the priority specified for their parent processes. Due to precedence constraints all event ends participating in the same event need to get the same value, which is equal to the highest priority present in any of the event ends. Thus, events ev3 and ev2 would get the priority of process P1, and event ev1 the priority of process P2. However, since a precedence constraint ev1->ev2 exists, the priority of the event ev1 needs to be readjusted in order to avoid priority inversion as described above in the procedure for realigning priorities of events. Thus, although preferred priorities of process P1, P2 and P3 are different, their execution pattern results in all events ev1, ev2 and ev3 having the same priority. The event-based scheduling approach uncovers realistic, priority-inversion free, values of priority levels, achievable with the given design of synchronization pattern between processes.

The procedure is not so convenient for application in systems with lot of recursions. There are two types of recursive processes: time-triggered recursion and ordinary recursion. In ordinary recursion there is a cycle and as a result all the events in the process have the same priority. The time-triggered recursions are considered new instances of tasks with new deadline values and there is no need to perform a circular update of priorities.

2.3.2 Equivalent automaton

From the discussion in Section 2.1.4, it is clear that assigning priorities to processes does not reduce the set of possible traces. Thus, one reasonable approach for checking real-time guarantees is treating CSP processes as automata and systematically replacing every composition of CSP processes with an equivalent automaton, and associating execution times and deadlines with points in the control flow of the equivalent automaton. In [11], it is in fact stated that timed CSP descriptions are closed-timed epsilon automata.

In order to simplify reasoning, in this paper we restrict the analyzed models to be free from the non-deterministic and too complex primitives: Systems are considered to be free from the usage of the *internal choice* operator and of those cases of the *external choice* operator that cannot be reduced to the *guarded alternative* operator. Internal choice is normally used as an abstraction vehicle and as such, it does not exist in final designs. External choice that cannot be replaced with a guarded alternative operator is another situation that is rarely used in practice and difficult to implement and also not straightforward to describe in automata representation. The decision here is to restrict final designs to be free of those two special cases. If the set of CSP operators is restricted in this way, the processes constructed using events and this restricted set of operators can be reasoned about using classic automata theory.

Automata theory [12] defines how to make a parallel composition of two automata. The example for this procedure is depicted in Figure 22. The start state of the equivalent automaton representing the composition is the combination of the initial states of the composed processes. In Figure 22, process P1 can initially engage in event a and process P2 in event b. Since event b must be accepted by both P1 and P2, only event a is initially possible. Event a will take the first automaton to state 2, while the second automaton will stay in state 1. Thus, starting from the initial state (1, 1) and following the occurrence of the event a, the composite state (2, 1) is discovered (see Figure 22).

For every reachable composite state all possible transitions are checked (taking into account when synchronization is required and when not). The resulting composite states are

mapped to the equivalent automaton. After a while all transitions either lead to the already discovered composite states or to the end state (when both participating processes are in their end states) if any.

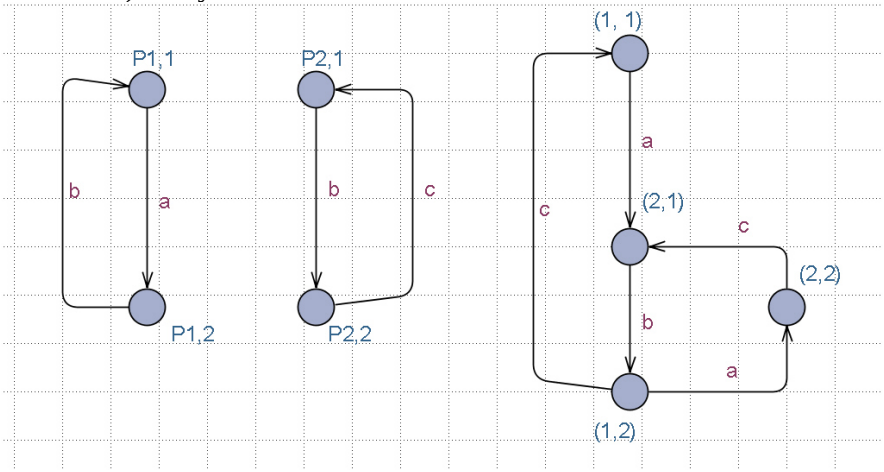


Figure 22 Construction of equivalent automaton

Some composite states are not reachable and thus not part of the equivalent automaton representing the parallel composition. In principle, composing, in parallel, a process containing 3 states with a process containing 4 states, yields a process with all 3*4 combinations possible. This is in fact the case whenever processes are composed in interleaving parallel. In non-interleaving parallel constructs, due to the involved synchronizations, the number of states is less.

A parallel composition can be seen as a way to efficiently write down complex processes that contain a number of states. Seen in that light, the introduction of the Parallel operator allows decomposing complex processes on entities that are smaller, focused on one aspect of the system at hand and simpler to understand.

The definition of the parallel operator is in CSP identical to the one in automata theory. The external choice of CSP viewed as automata is equivalent to making a composite initial state that is offering the set of initial transitions leading to the start states of the involved subprocesses and subsequently behaving as those subprocesses. The sequential composition is trivially concatenating the involved automata. If every CSP process is viewed as an automaton, creating an equivalent automaton representing a complete CSP-based application is a straightforward thing to do. The equivalent automaton defines all traces (sequences of events) possible in the system. Thus, it can be used as a model against which one can check different properties of the system – e.g. checking for deadlock/livelock freedom, checking the compliance of implementation to the related specification (refinement checking). For instance, an application has a potential for a deadlock situation if there is a state (not the end state) from which no transition is leaving. The refinement checking is about testing if a set of traces and failures defined by an automaton representing some implementation is a subset of the set of traces and failures defined by an automaton representing the related specification.

In Figure 23 SystemCSP based visualization of the equivalent automata from Figure 22 is given. The main difference of a SystemCSP representation compared to the automata way of visualizing CSP processes is that instead of on states, the focus is on events. The procedure of constructing an equivalent automaton is exactly the same. Focusing on events ensures that traces are more easily observable, especially if in SystemCSP, instead of the lines going back to the revisited states, as is always the case in automata, usage of recursion labels is enforced. Systematic usage of recursion labels will naturally separate subtraces

that are repeated and thus create immediately observable trees of possible event traces. Inspection of Figure 23 shows that possible traces in the single 'Loop' iteration of the equivalent automaton are <a,b,c> and <a,<b, a, c> ^{n},b,c>. The actual trace taken is dependent on the readiness of the environment. Recursion labels define sequences that are repeated and the IF choice and guarded alternatives divide traces into several subtraces.

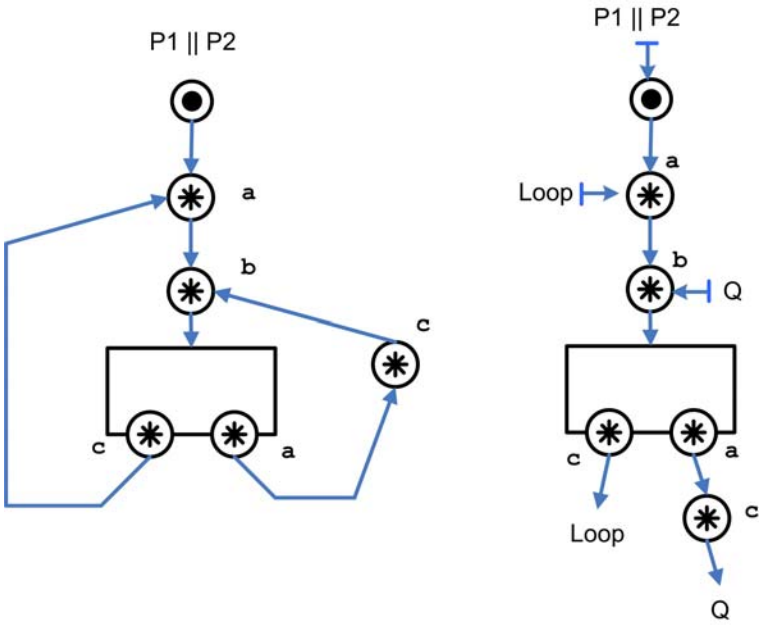


Figure 23 SystemCSP with recursion labels makes traces more obvious

2.3.3 Mapping time properties to equivalent automata

The next step is to extend the description of an equivalent automaton with time properties in a way that it will allow us to perform efficient analysis. The idea is to extend CSP descriptions with time properties in such a way that the mapping to the equivalent automaton preserves their meaning. The execution times of the calculation blocks can also be seen as related to the event ends immediately preceding them, that is, to event ends associated with events whose occurrence will allow every participating process to progress for the amount of execution time spent on the next calculation block.

The execution time of a process at a certain point of its execution is the sum of execution times along the path that brought the process to the current point. In other words, it is the sum of progress (expressed in time units) allowed by all event ends along the trace that process is following.

After specifying execution times and time constraints in the two subprocesses composed in Parallel, time properties are mapped to the equivalent automaton. If that can be done, then the analysis of time behaviour can be performed on the constructed equivalent automaton. If we are able to map the time properties from a pair of parallel composed processes to their composition, then the same can be done hierarchically in bottom-up manner yielding at the end an executable timed model of complete application.

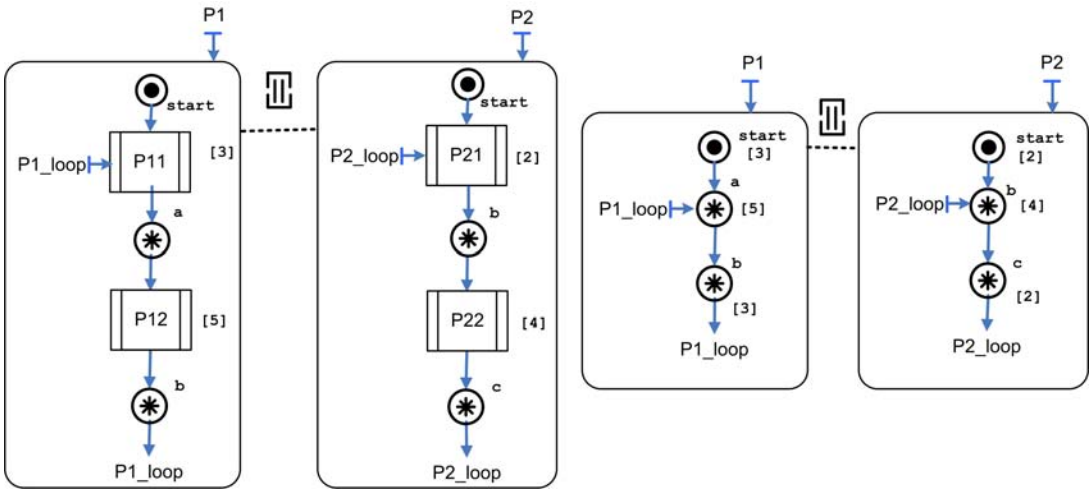


Figure 24 Specifying Execution times of code blocks

Analysis of time properties should be performed without the need to perform code block calculations. In such analysis, code blocks are substituted with their execution times and sums of execution times along all possible system traces are inspected with respect to the specified time constraints.

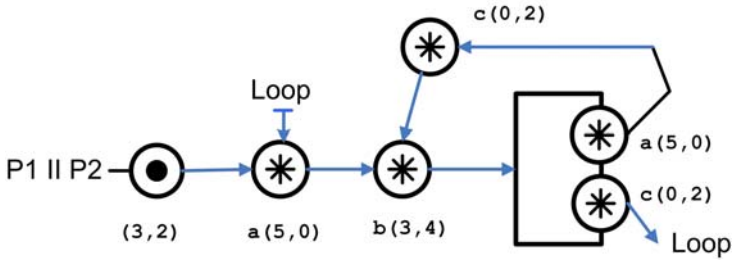


Figure 25 Equivalent Event Machine

The basic idea is that an event occurrence allows further progress of processes involved in that event occurrence. The initial event in process P1 (see Figure 24) allows process P1 to progress further in execution for 3 time units and offers event `a` to the environment. The initial event in P2 allows process P2 to progress in execution for 2 time units and then offer event b to environment. Thus, the composite initial event allows the involved subprocesses to progress for (3, 2) time units, where the first number maps to the event end in first subprocess and second number to the event end in the second one (see Figure 25). Event `a,` once it is accepted by the environment will allow progress of P1 for 5 time units and P2 for 0 time units since P2 is blocked waiting on its environment (including P1) to accept event b. Thus in the composite automaton, event `a` taking place following the initial event, will allow (5, 0) progress of the involved subprocesses. The subsequent occurrence of event `b` will allow progress of both P1 and P2, for 3 and 4 time units respectively, which is in Figure 25 expressed by associating ordered pair (3, 4) with the event `b`.

Note that in general, when a hierarchy of processes is resolved, it is not a good idea to capture progress of subprocesses as n-tuples. In a prospective analyzer implementation, since execution times are related to event ends, bookkeeping of allowed progress would be kept in the participating event ends and not in n-tuples, containing progress for all composed subprocesses. Different occurrences of the same event in a composite automaton

can in fact have associated different progress values. Essentially, execution times are expressed as the amount of progress event ends allow.

Note that the assumption here is that the environment is always ready to accept events. In fact, when the equivalent automaton is constructed hierarchically in a bottom-up approach, it will eventually include the complete system with all events resolved internally.

Unpredictable (in sense of time) occurrences of events from the environment can of course only be analyzed for a certain chosen set of scenarios. For every scenario, the environment can also be modelled as a process with defined time properties and composed in parallel with the application to form the complete system.

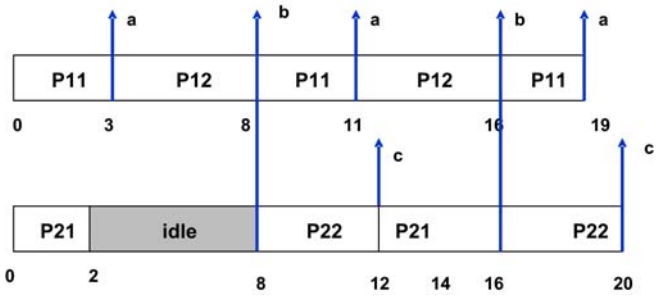


Figure 26 True parallelism

The actual time of event occurrences depends on the allocation. For the equivalent automaton of Figure 25, on Figure 26 true parallelism case is depicted, and on Figure 27 a shared CPU with P1 having higher priority and a shared CPU with P2 having higher priority. The same equivalent automaton keeps information necessary to unwrap actual timings of the involved events in all 3 cases.

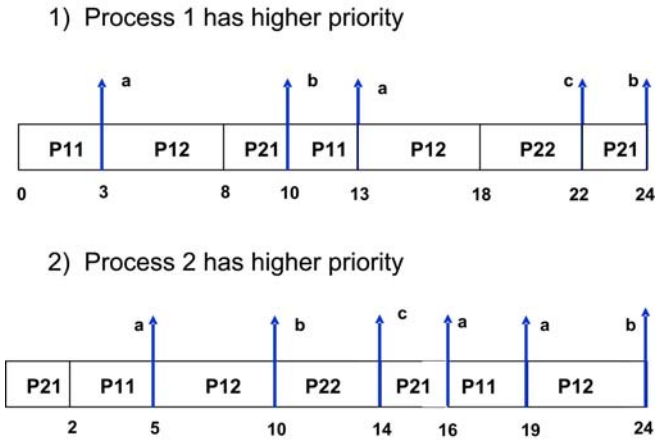


Figure 27 Shared CPU

In the case of true parallelism, components P1 and P2 are initially allowed to progress 3 and 2 time units respectively. Then event ‘a’ allows component P1 to progress another 5 units. Both processes synchronize on event b, meaning that their times must be same at the rendezvous point. Thus the time of this rendezvous point is max(3+5, 2+0)=8. Event b will allow components P1 and P2 to progress 3 and 4 units of time respectively. Under the assumption that the environment is always ready to accept events, event a will be accepted at time 8+3=11 and event c at time 8+4=12. This scheduling pattern is depicted in Figure 26. The scheduling pattern obtained for a shared CPU and different priorities of components is depicted in Figure 27.

Three independent timed models can be made by using in analysis either minimum, average or worst-case execution times. Using only the average execution time is a good first approximation of the system's behavior. Execution times are dependent on allocation of components to processing nodes and can in fact be measured or simulated for different targets and stored in some database. A prospective tool should be able to keep track of allocation scenarios and to simulate/analyze/compare effects of the different execution times in different allocation scenarios.

3. Conclusions

In this paper, ways to introduce time properties are defined in the scope of the SystemCSP design methodology. The specification of time properties is deduced by merging the ideas from previous work in the CSP community (Roscoe, Schneider). Implementation of CSP-based systems with real-time properties is then investigated. Two major directions are observed for achieving real-time: (1) introducing design patterns that can fit CSP-based systems into requirements of existing scheduling theories and (2) relying on constructing distinct scheduling theories for CSP-based systems. Comparing the two indicates is that the first proposed direction enables immediate implementation, while taking the second direction requires additional research. Thus, a recommendation for prospective tool used for editing SystemCSP designs is to use the combination of proposed design patterns and classical scheduling theories to provide real-time guarantees.

References

[1] Roscoe, A.W., The Theory and Practice of Concurrency. Prentice Hall International Series in Computer Science. 1997: Prentice Hall.
[2] Schneider, S., Concurrent and Real-Time Systems: The CSP approach. 2000: Wiley.
[3] Orlic, B. and J.F. Broenink. SystemCSP - visual notation. in CPA. 2006: IOS Press.
[4] Hilderink, G.H., Managing Complexity of Control Software through Concurrency. 2005, University of Twente.
[5] Buttazzo, G.C., Hard real-time computing systems: Predictable Scheduling Algorithms and Applications. The Kluwer international series in engineering and computer science. Real-time systems. 2002, Pisa, Italy: Kluwer Academic Publishers.
[6] Wittenmark B., N.J., Torngren M. Timing problems in Real-time control systems. in American control conference. 1995. Seattle.
[7] Boderc: Model-based design of high-tech systems, ed. M.H.E.U.o. Technology and G.M.E.S. Institute. 2006, Eindhoven, The Netherlands: Embedded Systems Institute, Eindhoven, The Netherlands.
[8] Buttazzo, G.C., Rate Monotonic vs. EDF: Judgment Day. Real-TimeSystems, 2005. 29(1): p. 5-26(22).
[9] Cervin, A. and J. Ekerz, The Control Server Model for Codesign of Real-Time Control Systems. 2006.
[10] Fidge, C.J., A formal definition of priority in CSP. ACM Trans. Program. Lang. Syst., 1993. 15: p. 681-705.
[11] Ouaknine, J. and J. Worrell, Timed CSP = closed timed epsilon-automata. Nordic Journal of Computing, 2003.
[12] Cassandras, C.G. and S. Lafortune, Introduction to discrete event systems. 1999, Dordrecht: Kluwer Academic Publishers.