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

Real-time and Fault Tolerance in Distributed Control Software[†]

Bojan ORLIC 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, NL-7500 AE Enschede, The Netherlands
E-mail: `B.Orlic@utwente.nl, J.F.Broenink@utwente.nl`

Abstract. Closed loop control systems typically contain multitude of spatially distributed sensors and actuators operated simultaneously. So those systems are parallel and distributed in their essence. But mapping this parallelism onto the given distributed hardware architecture, brings in some additional requirements: safe multithreading, optimal process allocation, real-time scheduling of bus and network resources. Nowadays, fault tolerance methods and fast even online reconfiguration are becoming increasingly important. All those often conflicting requirements, make design and implementation of real-time distributed control systems an extremely difficult task, that requires substantial knowledge in several areas of control and computer science. Although many design methods have been proposed so far, none of them had succeeded to cover all important aspects of the problem at hand. [1] Continuous increase of production in embedded market, makes a simple and natural design methodology for real-time systems needed more then ever.

1 Introduction

Being inherently parallel, control systems need transparent and structured way of using multithreading in a distributed environment. Formal checking based on CSP algebra has already been successfully applied to solve those problems in the *occam* programming language. This programming language was used for design and implementation of the scalable distributed systems constructed from *transputer* nodes and links. After the transputer disappeared from the market, several occam-like libraries [2],[3] were implemented in the popular programming languages. Section 2 starts by emphasizing the need to modify a way in which occam constructs are implemented in a distributed environment. Application model independent of target hardware architecture model is key to efficient and flexible design. However, in occam those two are not totally independent since not all constructs can be distributed. Constructs can actually easily be adapted to distributed form, if they are treated in the same way as all other process. They should communicate with their subprocesses using only channel communication and not knowing their identities. Afterwards, applicability of the CSP based framework in control is explored. In Section 3, various scheduling techniques for distributed occam-like programs are analysed. In Section

[†] 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.

4, the proposed approach to scheduling is extended to additionally lead to graceful degradation in case of processor overloads and network congestions. Fault tolerance issues and possibilities of system reconfiguration through mode changes are also discussed.

2 CSP application model

Hard real-time control data is usually exchanged over serial fieldbus protocols. The proposed distributed framework is an attempt to conceptually extend the occam-like library made by Hilderink [3], to achieve fault tolerance and real-time guarantees in fieldbus-based distributed control systems. Occam programs are organized as a nested hierarchical composition of constructs and processes. Nesting is allowed by the fact that constructs themselves are also processes.

2.1 The application model should be independent of the hardware architecture model

Separating application and target hardware architecture models of embedded system plays a key role in establishing a high degree of modelling and exploration flexibility [4]. An application model is independent of the target architecture if an application designer can make it exclusively based on the required functionality. Later in the design phase, this application model can be mapped onto the range of hardware architectures. Process allocation can be proposed in order to achieve real-time guarantees, best or fair utilization of all processors and communications.

Occam models are claimed to be architecture independent, but this is not quite true because only the parallel construct can be distributed (PLACED PAR). The actual design consists of several hierarchies of CSP constructs and processes that will run in parallel on different nodes.

The most desirable application model is the one where any process (keep in mind that constructs are processes too) can be deployed on any node. Thus all constructs should be able to be distributed in same way as a PLACED PAR from occam. Actually, this is true for almost all constructs. The PRIPAR construct was derived to assign priorities in using a basic shared resource - microprocessor. In a single processor system, PRIPAR priorities will indeed determine the order of execution. However, in a distributed system the role of PRIPAR construct is questionable and it does not make much sense to use it in the usual way. Besides easier readability, maintainability, and reconfigurability, purpose of using the SEQ construct, instead of putting whole sequential behavior inside one process, is so far an unused opportunity for distribution. Different parts of sequential behavior can be executed on nodes where the needed resources are. Sequential behavior can be implemented by using parallel processes, or in this case processes on different nodes, where the end of the first process in sequence triggers the next one and so on. Distributed SEQ organized in this way should not be confused with a *pipeline*, because the first process in sequence can be triggered again only after the last process in the sequence finishes. The ALT construct seems to be the most problematic for distribution. Guards can be attached to remote channels in same way they are attached to internal channels. There is just the additional communication overhead when remote guard becomes ready.

2.2 Constructs are processes too

In practical implementations of occam-like libraries, constructs are often not realized in the same way as other processes and instead of communicating only through channels, function calls are used. This creates a situation in which a parent construct holds pointers to its subprocesses, and each of them in turn has a pointer to its parent construct. When constructs are distributed over several nodes, this hardcoding is the source of many difficulties and artificial solutions must be applied to solve them. Furthermore, the essential reconfiguration power of CSP lies in the fact that processes do not know about identity of processes they are communicating to. Consequently, parent constructs should be protected from knowing the identity of its subprocesses.

A solution is to view the constructs only as processes that take care of the execution of a group of processes. Being a process itself, construct should communicate only through channels. Instead of holding pointers to each other, process and its parent construct need just to be correctly connected through a pair of channels. Thus, instead of using dedicated function calls, every process execution could be triggered over a channel. Similarly, the end of process execution is an event important for its parent construct. Instead of calling the parent construct's function, a process should use a dedicated channel to notify its parent about that event. Since each channel communication is also a potential scheduling point, this solution will be more preemptive in a natural way, eliminating the need for time-slicing algorithms. Not only is this approach more in the CSP spirit, but it will also allow easier distribution of processes over remote nodes. For instance a SEQ construct can trigger its subprocesses in sequence without knowing their location. Less flexible, but more efficient solution is to let each subprocess from a sequential construct, directly triggers next subprocess after it has finished. A distributed PAR construct can trigger all of its subprocesses on remote nodes using some kind of broadcast message and wait till all of its subprocesses finish their executions.

Also to dynamically update the list of nested processes, channels for adding and removing processes from/to a construct can be added to its interface. Those channels would be checked for changes on beginning of every execution cycle.

2.3 Applicability of CSP application model in control

In existing control systems, there is often a gap between timing constraints of the control theory model and its practical software implementations. This time inaccuracy can lead to uncontrollable performance degradations [5]. Control system can encompass many control and supervisory loops. Each loop can be further decomposed into the set of mutually dependent cooperating tasks, described using *precedence constraints* (see Figure 1).

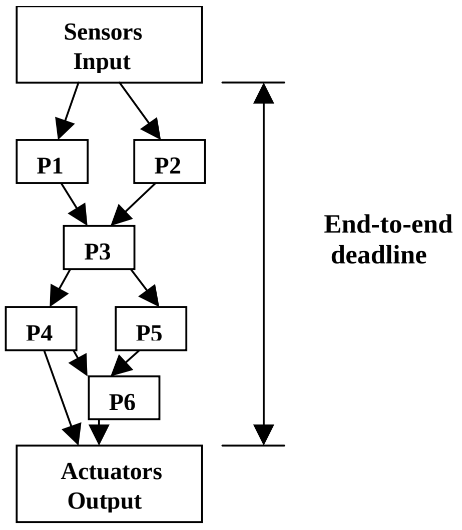

Figure 1: Possible precedence graph for one of control loop chains

Tasks are possibly distributed over different processors, near data sources they use or because some of them need special processors or hardware. Control loop chain starts with sensor data measurements and finishes with delivering command data to actuators. The order of task execution is on each processor determined by the use of an appropriate scheduling strategy. Scheduling requirements are determined by classifying tasks and messages, according to changes in their *utility functions* over time, in several well-known categories. *Hard real-time* tasks require *guarantee* to be given that it will finish its execution before *deadline*. In case of *firm real-time* tasks, there is no point to continue the task execution after missing a deadline. However their execution can occasionally be skipped without disastrous consequences. For *soft real-time tasks* reaching deadlines do not have to be guaranteed and applying only *best-effort* strategy is sufficient. *Non-real-time* tasks have no deadline. This classification sorts tasks based on the importance of achieving deadline. For control systems, except mentioned deadline-oriented tasks, especially important are *precise* [6] or *time-bounded* tasks [7]. *Precisely periodic* task [5] must execute, rather than just to be released, exactly one period apart in predefined time moments. Due to finite clock precision and the time needed for a processor to perform other activities, a small amount of jitter is unavoidable. Larger the jitter less is the utility obtained by task execution (see Figure 2). To reduce influence of other tasks on the size of this jitter, genuine precise tasks should not synchronize on events other then time events.

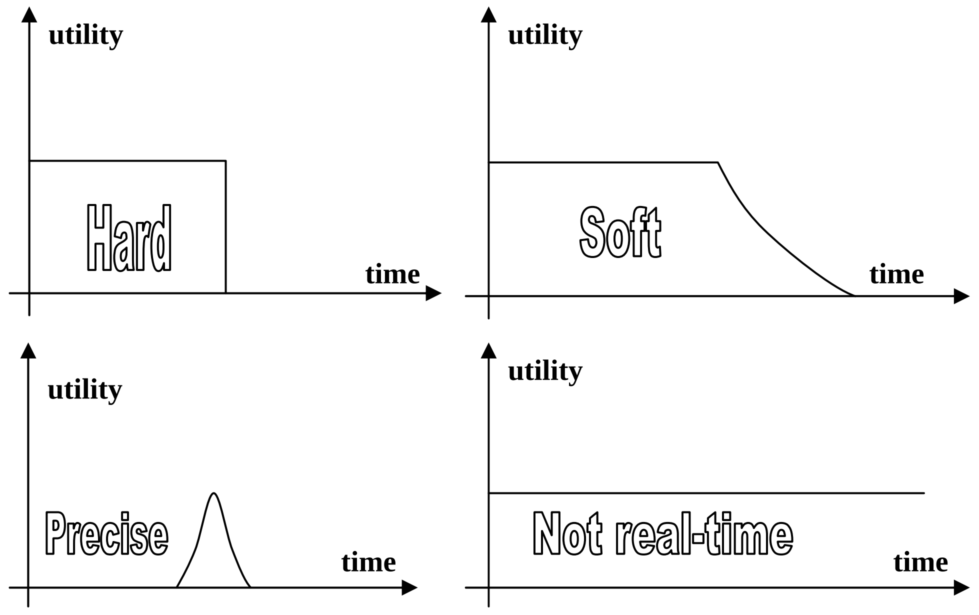

Figure 2: Utility functions for different tasks and message types (adapted from [6])

Although modern control theory is making some advances towards control algorithms that handle time varying delays, most of the contemporary control systems are based on control theory that assumes constant control delays and equidistant sampling [5]. Translated to the task domain, this means that sampling and actuation tasks should be *precisely periodic*. Note that precise and synchronized execution is more important in case of sampling tasks then in case of actuation tasks. The system state variables change value over time subject to physical laws. Jitter in sampling time will thus introduce uncertainty into the measured value. Because an uncertainty is introduced even before control computations are done, jitter in sampling time is more dangerous then jitter in actuating time. Since only a finite amount of energy can be delivered to an actuator in one sampling period, in many cases a little late actuation will still be better then no actuation. This leads to the implementation of actuation tasks as processes which will be released after both input data from preceding computational processes has arrived and a predefined actuation delay time is about to expire.

Often, instead of specifying explicit deadlines to every intermediate task forming a precedence graph of computational algorithm, this behavior is defined only through *end-to-end deadlines* [8]. End-to-end deadline is the only hard deadline for whole chain of tasks inside control loop.

Precedence constraints (see Figure 1) of tasks from each control loop can easily be translated into a hierarchy of CSP constructs and processes. Instead of specifying whole construct hierarchies, the application designer can only specify relationships between processes for which relationships are known. A design tool can then derive relationships that stayed unspecified, like proposed in [9]. Non-uniqueness of this mapping leaves additional space for various optimizations during design process.

Achieving real-time guarantees is most important in implementation of control systems. Especially appealing sub area of research in CSP theory, is attempt to give CSP support for real-time [10]. This real-time CSP theory is still not sufficiently mature for application design purpose. Actually real-time CSP theory can only make time specifications. Execution times of processes will always depend on properties of employed processors and communication links and those are described in the hardware architecture model. It is hard to expect that CSP theory will be extended in such an extent to behave like a simulator of hardware properties and behavior. It seems wise to view real-time scheduling and formal checking for multithreading hazards as totally orthogonal problems. Real-time behavior of systems is target-architecture dependent and rely on appropriate priority assignment scheduling, while occam-like process/channels application models enable the CSP formal check. Applying formal checks can guarantee freedom of *deadlocks*, *livelocks* and *race conditions* during normal system operation. But after component failure, components waiting on communication with failed components will be deadlocked. This situation must be handled by using timeouts and other error detection mechanisms, and by invoking recovery mechanisms possibly based on redundancy.

In order to really simplify system testing and validation by using CSP formal checks, the use of asynchronous communication must be maximally restricted. Furthermore, all parts of CSP based library including scheduler and various communication link receivers & transmitters should be built using construct / process / channel model. Only in this way, it is possible to be sure that overall system will not deadlock.

3 The target-architecture model: Can scheduling achieve real-time guarantees?

Instead of just gluing a communication layer to existing single system architecture, design of distributed real-time systems should be treated in the holistic way. Scheduling in distributed systems is more complicated then scheduling in its single processor counterpart. Reason lies in the fact that processor time scheduling and network resources scheduling are not totally independent of each other.

3.1 End-to-end deadline scheduling in distributed environment

In a distributed system, scheduling decisions are made locally for each node and our hierarchy of dependent occam-like processes spans over many nodes. If processes communicate reasonably often, natural scheduling/descheduling on channel communication will implement approximately fair distribution of processor time. But in real-time systems, fair distribution of processor time would conflict the fact that not all processes have same importance of execution. Some processes contribute more to the system behaviour then others. Some have more stringent timing requirements. The hierarchy of constructs and process encompasses part of the knowledge on process ordering. More precisely it defines which processes are done in a particular order and which are executed in parallel. Creating hierarchy of dispatchers by attaching a dispatcher to each construct works fine in case of a single processor system. Applying the same concept in a distributed system is not so good idea, since each dispatcher would under its control have subprocesses that might be on different nodes. Hierarchy of constructs and processes cannot directly be used to guarantee real-time scheduling. Still, this information reduces non-determinism considering the order of execution on each processor by imposing constraints on valid schedules. Those constraints can be used to transform the end-to-end deadline of each control loop into conservative deadlines attached to each process. Let us designate *worst-case execution time* of any process as WCET and *minimal case execution time* as MinCET. One of possible mappings for a set of processes from Figure 1 to CSP hierarchy of channels and processes is: (P1||P2);P3;(P4||P5);P6. WCET, MinCET and utility functions derived for the application model of that control loop are shown in Figure 3.

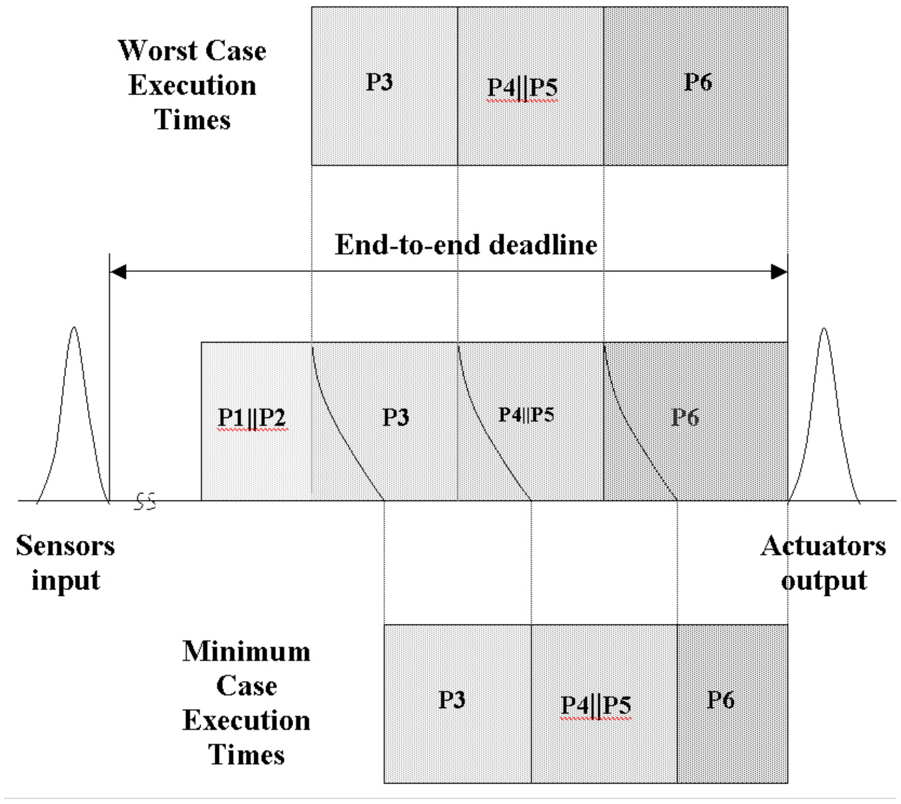

Figure 3: Deriving utility functions and deadlines for one control loop chain

According to [8] in case of a hierarchical approach to end-to-end scheduling, the best strategy is to use distributed scheduling based on both global and local information. In this case *global* information is obtained by offline use of application-specific constructs and process hierarchy to derive intermediate deadlines for each process. Those deadlines could then be applied *locally* for run-time scheduling. The *absolute deadline* is determined by subtracting ΣWCET of all following processes and remote communications from the time when a control loop chain execution cycle ends. If there are multiple paths through the chain for which real-time guarantee is necessary, the deadline should be calculated considering each path and the worst-case value should be chosen. The *relative deadline* is obtained by subtracting the task release time from the absolute deadline. The simplest and most often applied priority assignment method is *rate monotonic* (RM) approach.

3.2 RM approach

Complex control systems encompass several control loops with possibly different sampling periods. Those control loops might *not* be totally independent. For instance, supervisory loop can occasionally update a parameter inside some control loop. The rate monotonic approach would assign priorities to be proportional to the respective sampling frequencies. This gives the scheduler freedom to execute task anywhere between two consecutive periodic releases of the task. Note that this is somewhat different from *precise in time* execution. To guarantee real-time execution of all tasks, this approach assumes that scheduled processes are mutually independent and that deadlines are equal to the period, which does not hold in this case. Even on a single processor, RM approach can be strictly applied only if each control loop chain is implemented as one process with a deadline equal to its period. Applying rate monotonic scheduling locally on each node and on the network is obviously not identical to the case of applying it to the same software system on a single processor. Priorities are properties defined for each process on the system level, but applied on the local level. If we consider two active processes with different priorities on different nodes, process with lower priority on the less loaded node can be executed before process with higher priority on the overloaded node. Thus, in distributed systems global priority does not determine order of execution.

Even if we could prove schedulability of system designed in this way, this approach would still have a major disadvantage that it does not lend itself naturally to applying fault tolerance methods. Reason lies in the fact that in RM scheduling there is no distinction of importance priorities and deadline priorities. If we try to achieve graceful degradation by organizing a control loop process further as concurrent execution of several process hierarchies with priorities based on importance, priority inversion situation can happen. For instance let us consider two control loop chains with different sampling frequencies running in a RM-based PRIPAR. If the RM approach is used, RM priorities must have priority over importance priorities. Not important lowest priority optional process belonging to more frequently executed control loop, will have a higher priority then essential hard real-time process belonging to control loop executed less frequently. This example of priority inversion, resulting from mixing RM priority assignment and importance priorities, is illustrated in following occam-like pseudo code:

```
PRIPAR
..--Control loop 1
  WHILE TRUE
    SEQ
      TIME? samplingTime1
      ...Sensor measurments
      ...Control computations
      PRIPAR
        ...Essential (hard r.t.)
        ...Optional 1
        ...Optional 2
      ...Actuation
..--Control loop 2
  WHILE TRUE
    SEQ
      TIME? samplingTime2
      ...Sensor measurements
      ...Control computations
      PRIPAR
        ...Essential (hard r.t.)
        ...Optional 1
        ...Optional 2
      ...Actuation
```

3.3 *Deadline based scheduling*

Whenever the deadline is not equal to the period, modification of rate-monotonic approach known as *deadline-monotonic* (DM) priority assignment can be applied. Instead of getting priority proportional to the sampling frequency, each process will get priority inversely proportional to its relative deadline. The scheduler has now freedom to execute tasks in any time between task release and its deadline. In our case, conservative absolute deadlines can be derived for each process by subtracting the sum of WCET of all following processes from the end-to-end deadline. However, relative deadlines will depend on process release moments, which are dependent on the execution state of all processes in system and cannot be determined in practice. Strict application of deadline monotonic priority assignment is therefore not possible.

With absolute deadlines derived from hierarchy of constructs and processes, applying *earliest deadline first* (EDF) scheduling is possible. This approach is usually avoided because a large amount of overhead is introduced by recalculating priorities, especially in systems with several sampling frequencies. Introducing acceptable constraints can in some cases though lead to more efficient deadline and priority recalculation. Choosing sampling periods in such a way that there is not a to small *biggest common divisor* is usually not too much to ask. This value should not be small, because it will be the *basic period* designated by time reference messages, announced periodically over the fieldbus by a *time master*. Time reference messages must have highest priority on the fieldbus. If the maximum latency time for the highest priority message is not small enough, other ways (like interrupts) must be used to synchronise clocks. Local schedulers will use these messages to perform sampling and to synchronize their timers.

To efficiently use an EDF approach a *least common multiple* of applied sampling periods should be calculated as well. This value will designate the *large cycle* used as time line for expressing deadlines. Deadlines have not to be recalculated all the time. The scheduler knows the current time and the deadlines of processes expressed as offsets from start of the large cycle. A process only needs to recalculate its deadline at the very end of its execution by increasing the last absolute deadline for one period of its control loop. When the deadline value exceeds the large cycle, it is given its first value again. The scheduler always chooses to execute process with deadline nearest to current time. Thus, due to the special properties of control systems, recalculation of absolute deadlines for EDF scheduling can be done in very efficient way.

Advantage of using fieldbus time reference message compared to using local timers is that all sampling will be triggered simultaneously and with a time precision dictated by the most precise processor in system. As explained in section 2.1., synchronized and precise execution is more important for sampling then for actuating tasks. Therefore, while time reference message should be used to determine sampling points, time points of actuating can be determined using local timers. To ensure that sampling is done precisely in time and that sensor measurements can efficiently be distributed to all processes that need it, sampling processes should not synchronize with computational processes directly. Same sensor data, needed by several processes on different nodes, can be broadcasted. Synchronization and triggering control loops can be done in separate *trigger process*. This process execute triggering scheme describing the time execution pattern of the current operation mode. The trigger process is executed after all required sampling processes have finished or after the timeout in case of sensor failure. Thus it can also handle cases when various sensors data is unavailable. To minimize communication overhead trigger process is distributed. On each node part of triggering scheme, concerning only the control loops initiated there, is placed.

To illustrate some of architecture decisions more clearly simple control system will be considered as an example. Let system consists of Control loop 1 and Control loop 2 with sampling periods T and 2T respectively, and Supervisory loop executed every 3T, where T is length of the basic cycle. Required length of one large execution cycle is obviously equal to 6T. More insight into the time patterns of loop executions can be given using some kind of sketch as in Figure 4. The vertical axis shows the flow of time inside one large cycle and the horizontal arrows present points in time when precise sampling and actuation tasks should be executed.

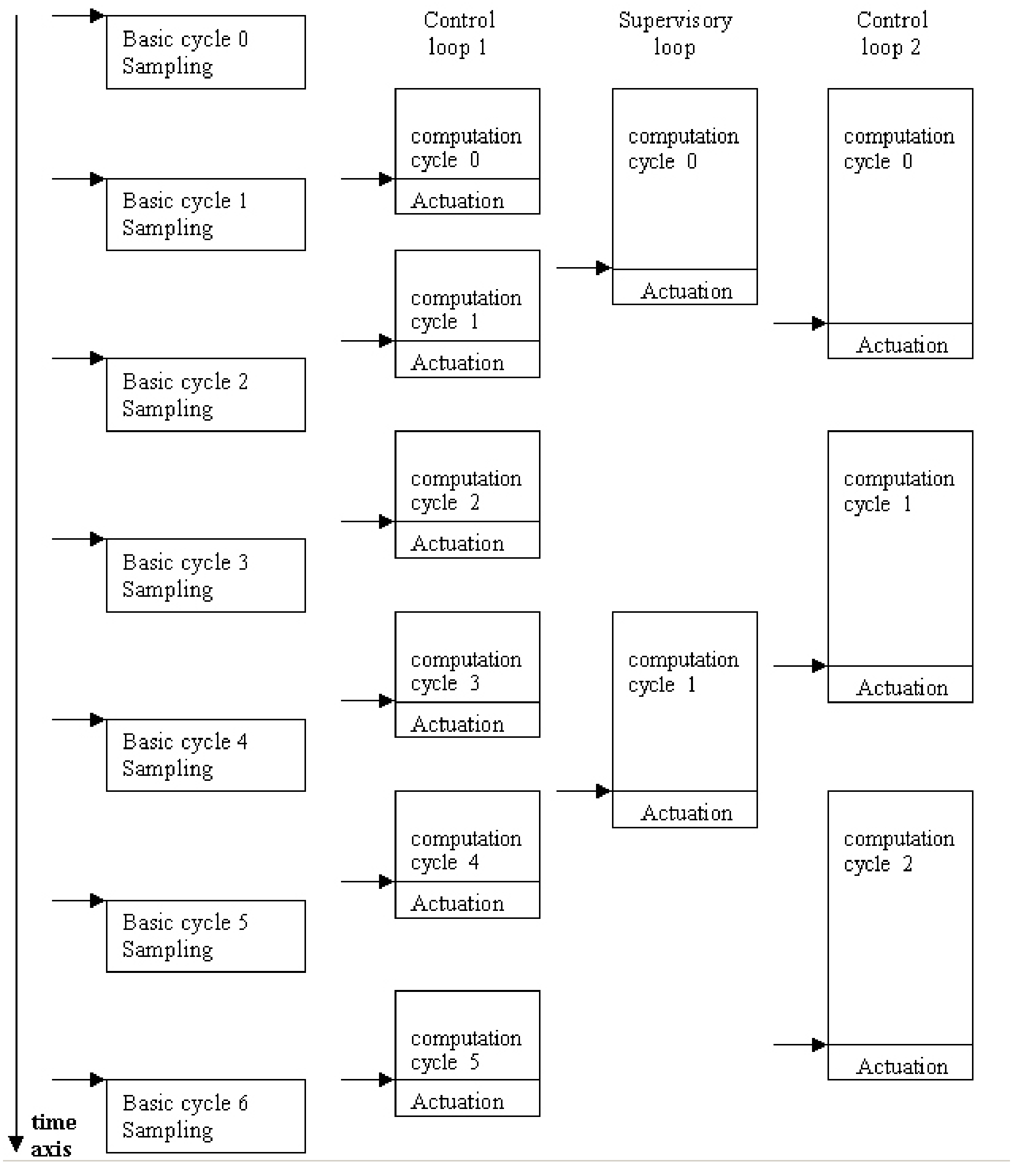

Figure 4: System consisting of three control loops executed on different sampling frequencies

A readable way to describe the functionality of the highest level of the system is using some kind of occam-like pseudo code:

```
largeCycleSize=6
Interrupt Service Routine SAMPLING_ISR[i] {Signal data[i] ready}

PAR
  SEQ
    WHILE(true)
      if((basicCycleNum++)==largeCycleSize) basicCycleNum=0
      FIELDBUS_Broadcast? time ref. msg. {largeCycleNum, basicCycleNum}

      -- after receipt of this message on each node START_CONVERSION(basicCycleNum)
      -- process will trigger sampling of appropriate A/D converters based on
      -- basicCycleNumber; after conversion completion interrupts are triggered

      START_CONVERSION(basicCycleNum)
        CLOCK? SamplingTime
        PAR
          SamplingProcess[0]
            -- wait data[0] ready
            A/D_Conv_register[0]?data[0]
            OVERWRITE_BUFFER[0]!data[0]
          SamplingProcess[1]
            -- wait data[1] ready
            A/D_Conv_register[1]?data[1]
            OVERWRITE_BUFFER[1]!data[1]
          ...
      -- This PAR construct on exit broadcast SamplingOver message

      --Trigger process
      FIELDBUS_Broadcast? SamplingOver{list Of unavailable Sensor data}
      -- here unique triggering scheme is shown. When not all control loops
      -- start on same node, triggering is done only on node where loop starts
      SWITCH (basicCycleNum)
         CASE 0:
           Control_loop1.START!SamplingTime
           Control_loop2.START!SamplingTime
           Supervisory_loop.START!SamplingTime
         CASE 1:
           Control_loop1.START!SamplingTime
         CASE 2:
           Control_loop1.START!SamplingTime
           Control_loop2.START!SamplingTime
         CASE 3:
           Control_loop1.START!SamplingTime
           Supervisory_loop.START!SamplingTime
         CASE 4:
           Control_loop1.START!SamplingTime
           Control_loop2.START!SamplingTime
         CASE 5:
           Control_loop1.START!SamplingTime

  --Control_loop 1
  SEQ
    WHILE(true)
      START? SamplingTime
      PRIALT
        PARAMETERS1_UPDATE?par
          parameterUpdate()
        SKIP
      ...Get sampled data from overwrite buffers
      ...CALCULATIONS (hierarchy of distributed CSP processes and constructs)
      CLOCK? AFTER SamplingTime+ControlDelay[1]
      ...ACTUATING
  --Control_loop 2
  SEQ
    WHILE(true)
      START? SamplingTime
      ...Get sampled data from overwrite buffers
      ...CALCULATIONS (hierarchy of distributed CSP processes and constructs)
      CLOCK? AFTER SamplingTime+ControlDelay[1]
      ...ACTUATING
  --Supervisory_loop
  SEQ
    WHILE(true)
      START? SamplingTime
      ...Get sampled data from overwrite buffers
      ...CALCULATIONS (hierarchy of distributed CSP processes and constructs)
      ...PARAMETERS1_UPDATE!par
```

`START_CONVERSION` is dependent of `basicCycleNum` value because not all sensor measurements are performed every basic cycle. This variety of sampling frequencies reduces the possibility to trigger all A/D conversions using a single broadcast signal. A satisfying alternative is to implement this part of code as a short sequence of assembly lines, triggering conversions always in the same order and with the same offset within sampling period. Once the measured data is collected in the registers of A/D converters, the execution order of sampling processes fetching that data is not so important. All those processes can be executed in parallel. Sampling processes that do not require data every basic cycle will have an internal counter to effectively reduce the sampling frequency. Same sensor data can be used by several processes from different control chains. Some of those control loops might have an execution frequency less then the sampling frequency of the used sensor. The sampling process might in that case write data to an overwrite channel or buffer in case all samples should be collected.

Each calculation block represents a hierarchy of distributed mutually dependent processes. Each of those processes has an initial deadline value calculated offline based on WCET times and the end-to-end deadline. On the very end of each process execution, its deadline is recalculated by adding the period. Obtained deadlines are used in local processor and network schedulers to implement EDF policy.

For fault tolerance reasons, several nodes equipped with high precision oscillators are capable to take over the role of the time master in case it fails. If some process misses its relative deadline, this still does not imply failure of the system. Reason is that extra conservatism is added while transforming end-to-end deadline into deadlines of intermediate processes. It is important to perceive that only the end-to-end deadline of the whole control loop chain is a hard deadline. If any subprocess misses its deadline, derived under worst-case assumptions, that still does not imply that end-to-end deadline will also be missed. Deadlines derived for intermediate processes can thus be classified as soft real-time, as it was depicted in Figure 3 using concept of utility functions. From Figure 3 it is possible to conclude that missing end-to-end deadline is unavoidable only after the current process is not completed and time reaches value determined by subtracting ΣMinCET of all following processes from the end of the loop chain cycle time. At this point, an exception is raised and the system performs what ever fault-tolerance measures are needed to prevent disaster [11].

4 Graceful degradation and reconfigurability

Being mission-critical by definition, hard real-time control systems usually have to encompass some support for fault tolerance. Sometimes, perhaps due to failed components, there is not enough processor time available to perform all computations or not enough network bandwidth to send/receive all messages in time. In same time it is important to observe that not all tasks and messages contribute equally to the overall system behavior. Instead of complete failure, reaction to processor overloads or network congestions must be omitting execution of less important processes resulting in somewhat lower, but still acceptable performance of delivered service. Reconfiguration is introduced due to hybrid nature of the designed system or due to the need for software restructuring as respond to, planned or failure caused, changes in hardware architecture or environment.

Several approaches to graceful degradation can be used:

1) Increasing sampling periods of control or supervisory loops can reduce time needed for application execution. This is possible because the sampling period is usually not determined based only on Shannon's theorem, but also to satisfy some performance requirements [5]. However, changing a sampling period implies that prefilters must be changed too [12]. Possibly, this can be implemented. Anyway,

filter dynamics and sampling periods are parts of system model and thus the whole model must be recalculated. [12] A solution could be to predefine alternative computational algorithms for several values of sampling periods.

2) Simplified alternative processes that need fewer resources can be employed, resulting in degraded quality of service.

4.1 How can graceful degradation be naturally achieved?

The methods described in section 3.4 attach priorities directly to each process and leaves the PRIPAR construct unused. One of the reasons is that PRIPAR priorities can be defined only during design time, and deadline based priorities are changing dynamically. More structured approach would be to derive PRIPAR priorities based on the importance level of process (essential, optional level one, optional level two …). If priorities were derived hierarchically, their values would be dependent of application structure. Although nested hierarchies of PRIPAR processes exist, *importance priorities* should not be dependent of the position in the hierarchy. This means that process importance priority defined inside PRIPAR is also its global importance priority. Essential processes would have the highest importance priority and several additional levels of optional processes can exist in the system. Thus, the lowest importance priority that can be executed in time, will define the level of system performance. Assigning this kind of priorities can help to divide system execution on several layers and accomplish graceful degradation in case of overload. If any essential process needs to be executed, it will always have priority compared to other optional processes. If the system is additionally made as totally pre-emptive, processes from a lower priority level can never jeopardize real-time execution of higher priority layers. Thus, the essential layer can be seen as separate hard real-time system. Several optional levels with high priority would form the soft real-time layer of the application and the rest of the levels would not have deadlines and will form the non real-time layer of the system. Execution of firm processes can be omitted occasionally, but not every time. To prevent starvation, firm process can, in addition to its basic importance priority, experience temporary priority boost each time it is omitted. This boost is cancelled after such a firm process is executed. To keep guarantees for the hard real-time layer unaffected by this, importance priority of firm process after applying priority boost must always be less then the essential priority.

The deadline value and importance priority are encapsulated in each processes, and made viewable to the scheduler. The schedulability is tested offline during the process allocation phase. Some processes will have a fixed allocation due to their connection to specific hardware. Allocation of the other processes can be determined in a way that will guarantee real-time behaviour of overall system. The allocation is the result of a complex optimization process aiming at fair distribution of the load to all nodes and minimizing network communication, especially for high priority processes, while satisfying time constraints. Local schedulers are executing EDF policy with several priority groups according to graceful degradation levels.

4.2 Graceful degradation in the communication layer

Absolute deadlines can also be derived for each remote communication. Priority of messages on the fieldbus will be determined based on the importance priority and its deadline. Dominant part of message priority is the importance priority, which is set to the value of priority of the sending process. In this way, more essential layers of execution will always be given priority over more optional layers. This means that in case of *network congestion*, the ability to transfer messages in time will gracefully degrade. The second part of the message priority will be derived based on its deadline if there is one. For instance, in case of CAN, the message ID is used both for arbitration and addressing purposes. Such a CAN message ID can, therefore, be divided into several fields sorted by relevance: importance priority field, deadline priority field, one bit for specifying whether data or protocol control information is transmitted, one bit for specifying if content is rendezvous message acknowledgment or plain message, one bit for broadcast and two fields needed to uniquely identify each channel: node ID field and channel ID field. By using node identifiers Channel IDs can be assigned locally on each node.

Priority(importance|deadline)	Protocol/data	/ack	broadcast	nodeID	channelID
(PRIORITYFIELDSIZE)	(1B)	(1B)	(1B)	(NODEIDFIELDSIZE)	(CHANNELIDFLDSIZE)

Main principles of communication subsystem are to encapsulate package protocol details in a passive link driver[13] object and all hardware specific functionality in a passive device driver object. Link drivers are plugged into the remote channels transparently of user process. In this way, the process using a channel does not know whether that channel is remote or local. Each remote channel has a link driver object and there is one unique device driver object per node. Any two link drivers speaking the same language (having same package protocol) can communicate. Interrupt service routines initiated by the hardware side of communication subsystem will only release blocked processes: producers or consumers to allow them to continue executing write () or read () function of link driver. Synchronization to access the communication resource (like CAN transmit mailbox) as well as all other device specific functionalities and register configurations are hidden in the device driver object. The prioritization in accessing communication resources is done in a natural way because semaphores sort waiting-queues based on priorities of blocked processes.

4.3 Reconfigurability

Many control systems are *hybrid* and encompass several *modes of operation* depending on current place in the system’s state-space. For instance, in airplanes separate modes can be defined for: take-off, normal flight, landing and various emergency situations. In systems with several modes of operations, each process can be used in one, several or all modes. In each mode, the same process can be employed inside different configuration and with different priority level. For every mode, separate hierarchy of constructs is predefined. Only one of several alternative hierarchies can be active at the same time. Thus, mode change will trigger fast system reconfiguration by choosing an alternative hierarchy of constructs to be executed in the next cycle. Information on the current mode or information on the mode change must still be passed to a process whenever it is triggered. This is another situation in which the described solution using start channels to trigger process execution from its parent construct appears to be useful. In each mode, different levels of graceful degradation are defined by designating different priorities to system parts. During the design phase, each mode must pass a schedulability test, a CSP formal check and a control system performance simulation. Once the system is designed in this way, a central reconfiguration management process (possibly being implemented as human operator) can only trigger mode changes, which will be performed automatically.

More flexible online reconfiguration can be achieved by implementing a reconfiguration tool on a remote node. This tool would monitor program execution and perform optimization techniques for load balancing and it would execute schedulability tests and formal check all potential alternative configurations. Due to the design principle, described in section 3.1, that processes are not hard-coded into the parent constructs, hierarchies of constructs and processes could be modified easily for inactive modes. New modes could also be defined, tested and loaded online. Such software system would be able to respond without restarting to major planned changes in software as well as in its environment structure. New nodes could be added and employed easily; nodes that need maintenance would be removed from system through planned reconfiguration and possibly transient period of system performance in appropriate level of graceful degradation.

4.4 Fault tolerance issues

Fault tolerance is important aspect of mission critical control systems, and it must be considered during design of such a system in as much as possible natural way. Reliability of control systems is usually increased by choosing robust control laws, non-sensitive on order of operations or finite precision of calculations, or by calculating and reducing the sensitivity of control loop on time delay, inserting redundant sensors or actuators into system, using sensors to check if actuators had performed requested action [12]. But besides this design time approach, additional software methods are needed to enable run-time error detection and determine how the system should respond in case of error. Therefore, the next research area would be to investigate possibility and adjustments needed to apply well-known fault tolerance techniques in the context of this framework. Refining every PAR construct to PRIPAR, by defining importance priority of each process, will divide the system into several layers of graceful degradation. But this is not enough. Fault tolerance must be taken care at all levels in the system architecture: on level of signals, data structures, processes, channels, constructs, nodes, network, cluster of nodes and on global system level. On each level, appropriate error detection and recovery mechanism can be applied and a minimum level of service is defined to distinguish operational and failed components.

Every primitive process is sequential and every sequential execution can easily be divided into one or several *recovery blocks*. Recovery block [1] including several concurrently executing processes is much harder to implement. After error is propagated through inter-process communication, all processes participating in that communication can be affected. Rigorous application of recovery blocks in primitive processes and sequential constructs will minimize the probability of error propagating through channels. However, this might *not* be enough, since acceptance test might need to check invariants including global state variables defined out of the scope of such recovery block. In a software system organized as hierarchical layers of communicated processes, the global state is distributed locally in every process. While sequential execution can be considered to change the state in deterministic way, concurrent execution yields different patterns of state traces, depending on actual order of execution. The correct point for executing acceptance test in PAR constructs is immediately after all inner processes in that parallel construct have finished execution and the final state of the current execution cycle is reached. One of possible implementations is that each process makes part of its internal state viewable via an asynchronous channel. If the acceptance test fails, all processes from the recovery block reload previous state. Acceptance tests must contain knowledge of state variables relations as well as which processes are maintaining needed state variables. Thus, those tests must be defined separately for each mode and redefined whenever some operation mode is redesigned. If processes from a PAR construct can be partitioned into groups with no communication between groups, separate recovery block can be defined for each group. Only for those groups that did not pass the acceptance test, alternative execution is started.

Processes or constructs must report their failures to their parent constructs using status messages in their end channels. This is another important factor in deciding to use end channels instead function calls to notify parent processes of finished execution. Service status messages can deliver combination of error flags: *value error* in case of failed acceptance test or *timeout error* in case of missed deadlines. If the detected error cannot be handled locally, it is propagated to higher levels in hierarchy. When the error reaches the highest level and cannot be handled, some emergency mechanism is used to get the system to a *safe state*. If the part (node, sensor, actuator, network) that needs maintenance is identified, the problem is solved by excluding that part from system operation and replacing its functionality by activating passive replicas on other nodes.

5 Design tool requirements

A designer must be supplied with a tool that offers a simple transformation path from complex requirements to a simple, safe and testable implementation. A control algorithm and its implementation represent two views of the same system model. Therefore, the strength of the used design methodology would especially be improved if the design tool is made in combination with simulation and control design tools like Matlab, Simulink or 20SIM. In those tools, system models and control algorithms are traditionally designed using a functional block diagrams. When translating this to CSP, each block could be turned into a process, and passing data between blocks would become communication over CSP channels. In the implementation phase, the tool would take as inputs this occam like structure derived from the control algorithms. Real-time requirements of model would be refined by specifying sampling periods and end-to-end deadlines for each control loop chain. To ensure that automatically generated code is safe, design tool must be equipped with a CSP-based formal checker. Combining this with some graphical modelling language, like GML proposed by Hilderink [9], would make the design process even more natural and easy. CSP models should include this library's underlying multithreading kernel model and network protocol models. The application level design methodology could be further improved by translating existing design patterns to occam-like application models and constructing new CSP design patterns as guide to solving typical problems in the real-time control application area.

The design tool should be able to recalculate reliability and other dependability features of the system. Refining every PAR construct to PRIPAR, by defining importance priority of each process, will divide system into several layers of graceful degradation. For any signal, the allowable range and maximum rate of change and additional ways of protection could be specified. A library of fault-tolerance enabled components and design patterns can be made to facilitate automatic code generation. For instance, replicated PAR components can be made to ensure replica's synchronization and consistency. Also based on given invariant constraints, a tool could propose how to construct recovery blocks and generate code automatically. Influence of various faults and component failures should be easily simulated, leading to discovery of weak points in system.

Architecture model should describe used hardware components. Programmable processing node models (memory model, instruction execution times, cache…) and detailed fieldbus models should be made. Using those models, time and reliability behavior of single application model could be simulated for range of target architectures. Based on those simulations and analysis, appropriate architecture and optimal process allocation would be chosen. It would be possible to validate complete system by refining application level simulation considering influence of chosen architecture. For instance, special libraries like TrueTime and Jitterbug [14] can be used to explore influence of determined worst case time-delays on control algorithm, giving us opportunity to adapt control algorithms.

6 Conclusions

In this paper, several design principles were proposed leading to consistent, designer friendly, CSP-based architecture for design of fault tolerant, distributed, hard real-time control systems. Avoiding hard coding of processes inside parent constructs leads to enhanced flexibility and easier distribution. The proposed priority assignment method integrates graceful degradation and real-time scheduling. This approach tries to make a unified view on several distinct important areas in real-time control software design. It expresses need to treat design of distributed real-time systems as whole and not as gluing real-time communication layer to existing systems. Properties of design tool needed to achieve this unified design view are explored.

References:

[1] Burns A., Wellings A., *Real-Time Systems and Programming Languages Ada95, Real-time Java and Real-Time POSIX*. Third ed. INTERNATIONAL COMPUTER SCIENCE SERIES, ed. M. A.D. 2001, Essex, UK: Pearson Education Limited.

[2] Welch, Peter H. *Java Threads in the Light of occam/CSP*. in *Architectures, Languages and Patterns for Parallel and Distributed Applications, WoTUG-21*. 1998. Amsterdam: IOS Press.

[3] Hilderink, G.H., Broenink, J.F., and Bakkers, A.W.P. *Communicating threads for Java*. in *Proc. 22nd World Occam and Transputer User Group Technical Meeting*. 1999. Keele, UK.

[4] A.D. Pimentel, P. van der Wolf, E.F. Deprettere, L.O. Hertzberger, J. T. J. van Eijndhoven, S. Vassiliadis. *The Artemis Architecture Workbench*. in *Progress workshop on Embedded Systems*. 2000. Utrecht, the Netherlands.

[5] Wittenmark B., Nilsson J., Torngren M. *Timing problems in Real-time control systems*. in *American control conference*. 1995. Seattle.

[6] Buttazzo, Giorgio 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.

[7] Bakkers A.W.P., van Rooij R.M.A., James L . *Design of a real-time operating system (RTOS) for robot control*. in *OUG-7 Technical Meeting*. 1987. Grenoble, France.

[8] Liu, Jane W. S. *Issues in distributed real-time systems*. in *IDA Workshop on Large, Distributed, Parallel Architecture of Real-Time Systems*. 1993. Fairfax, Virginia.

[9] Hilderink, Gerald H. *A graphical Specification Language for Modeling Concurrency based on CSP*. in *Proc. Communicating Process Architectures 2002*. 2002. Reading, UK.

[10] Schneider, S., *Concurrent and Real-Time Systems: The CSP approach*. 2000: Wiley.

[11] Orlic B., Broenink, J.F. *CSP channels for CAN-bus connected embedded control systems*. in *Progress 2002 Workshop*. 2002. Utrecht.

[12] Astrom, K.J., Wittenmark B., *Computer-Controlled systems : theory and design*. 3rd ed. 1997: Prentice Hall, Inc.

[13] Hilderink, G.H., Bakkers, A.W.P., and Broenink, J.F. *A Distributed Real-Time Java System Based on CSP*. in *The third IEEE International Symposium on Object-Oriented Real-Time Distributed Computing ISORC 2000*. 2000. Newport Beach, CA: IEEE.

[14] Cervin A., Henriksson D., Lincoln B., Eker J., and Arzen K., *How Does Control Timing Affect Performance?* IEEE Control Systems Magazine, 2003. **23**(3): p. 16--30.