CAN Fieldbus Communication in the CSP–Based CT Library

Bojan Orlic, Hany Ferdinando 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 – **In closed-loop control systems several real-world entities are simultaneously communicated to through a multitude of spatially distributed sensors and actuators. This intrinsic parallelism and complexity motivates implementing control software in the form of concurrent processes deployed on distributed hardware architectures. A CSP based occam-like architecture seems to be the most convenient for such a purpose. Many, often conflicting, requirements make design and implementation of distributed real-time control systems an extremely difficult task.**

The scope of this paper is limited to achieving safe and real-time communication over a CAN fieldbus for an existing CSP-based framework.

I. INTRODUCTION

CSP [1, 2] is a formal algebra that can be used to test multithreading software for undesired conditions like deadlocks, livelocks and starvation. Essential elements in the CSP theory are *processes* and *channels*. A primitive process is a piece of code executed in sequential manner. Processes can, however, be hierarchically organized using other processes as building blocks in *sequential*, *parallel* or *alternative* constructs. Communication between processes is only possible over special synchronization primitives called *channels*. Channels serve to hide the location and the identity of communicating processes from each other. This decoupling of processes yields possibility to change the program structure without making changes in the processes themselves, enhancing reconfigurability and reusability. Channels in CSP are *rendezvous*, meaning that communication takes place after both the producing and consuming process are ready for communication. Processes never access hardware directly and are therefore hardware independent.

CSP algebra has already been successfully applied as the basis of the *occam* programming language [3]. This programming language was used for design and implementation of the scalable distributed systems constructed out of *transputer* nodes and links [4]. After the transputer disappeared from the market, several occam-like libraries [5],[6] were implemented in popular programming languages. In our laboratory, Hilderink [7] has developed such a library (the CT– Communicating Threads –library) with versions in C, C++ and Java. This CT library differs from similar occam-like libraries in application area, the CT library is tailored for real-time embedded systems [8]. Instead of relying on different and often non-deterministic scheduling mechanisms of underlying operating systems, the CT library is made as small kernel performing thread scheduling. Distribution is based on the *link driver* concept [7]. In this concept, each physical link has an associated passive link driver object which encapsulates hardware dependent details of communication. All channels from the same node sharing the same physical link have plugged-in same link driver (see Figure 1).

The link driver concept frees processes of knowing whether a channel is remote or not, which strongly contributes to overall program flexibility and reusability. Furthermore, channels themselves do not know anything about the implementation details of real communication; they just use the link driver plugged in them, if any. Application programmers deal with an application model consisting only of channels, processes, composition constructs and allocation of processes on nodes. Actual implementation of distribution on various network links is hidden from the application programmer. Thus,

allocation of processes on processors can be changed without reconsidering the process' internal details. Portability is obtained in a sense of user level application. However, each new processing board demands building specific link drivers for every type of its links. Yet, the system can be considered open, because new link drivers can be designed and plugged into the system using the well-defined interface of the CT-library.

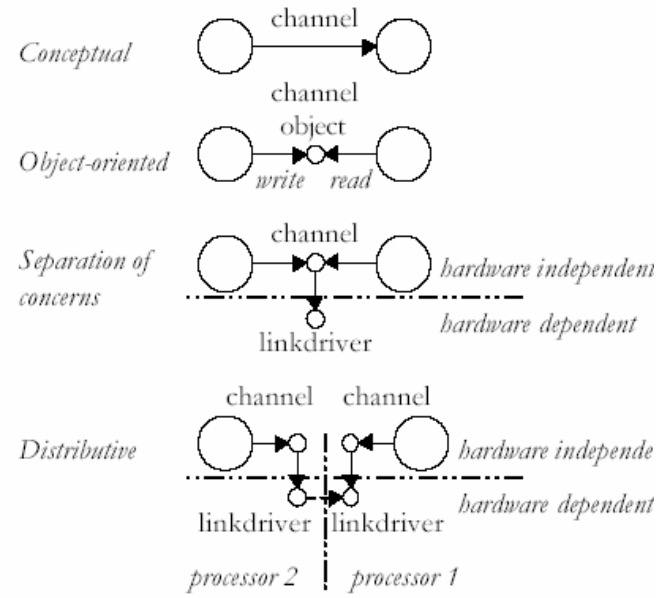

Figure 1: Channel Framework in the CT library

Of course the link driver concept is still not perfect. It has significant weak sides too. Reliability is not achieved since there is no mechanism to use alternative links when the originally assigned one fails. There is no notion of achieving quality of service requirements (like real-time delivery,..) in such a link drivers. Also no addressing and naming concept is associated with the existing link driver distribution model, because a point-to-point connection as in CSP and occam, is assumed.

Even if the system is initially made homogeneous, during its life-time it will evolve and in that process introducing heterogeneity is often unavoidable. Thus, it makes sense to design distributed systems with heterogeneity in mind from the very start. The existing CT-library framework solves this problem only considering hardware heterogeneity through the already explained link driver concept. Programming language heterogeneity is possible, since only pure data is communicated. Yet, data conversions are still necessary (i.e. the functionality of the OSI presentation layer), since different programming languages and compilers do not necessary always use the same data format (cf. the big Endian and little Endian formats). A link driver being common for all channels using the same link is not suitable for this purpose. Platform (i.e. operating-system) independency could be disregarded if only the CT library kernel is used. However some parts of complex systems (etc. logging and monitoring utilities) require more operating system services then pure thread management. Besides, the Java version of the CT-library requires a JVM and an operating system running. Finally, different channels should be able to implement different security policies. The place to implement security policies is in channels, and not in links. (Note that channels are the functional connections between processes and links are the physical implementations of channels crossing node boundaries.) To efficiently achieve all those requirements further modularization and decomposition of link driver model is needed.

This paper is a result of an attempt to improve the link driver framework based on experiences gathered during porting the CT library to a DSP-boards communicating over a CAN fieldbus link. An important issue is to determine the minimal set of modifications this library needs in order to enable real-time closed control loops over fieldbuses.

II. APPLICABILITY OF DIFFERENT DESIGN APPROACHES IN EMBEDDED SYSTEMS

In embedded systems memory is scarce resource. Embedded software is still optimized to have a small as possible memory footprint. Because of unpredictable execution times of memory allocation and the possibility for memory fragmentation while dynamically reallocating memory during system operations, in real-time embedded systems memory allocation is allowed only during the initialization phase.

In hardware implementations of most fieldbus system protocols, arrival of a package on a link causes an interrupt. Hardware devices are designed with the idea that Interrupt Service Routine (ISR) execution is more important then execution of any other software process. Thus the ISR will break any process being executed and run its code in context of the interrupted process. During its execution, the ISR can be preempted *only* by other ISRs. To eliminate the possibility that ISRs will delay execution of possibly more important processes, they are written to be as short as possible. Therefore, no significant memory allocation is allowed inside a ISR. Furthermore, preemption and context switching inside a ISR is most often too complex to implement correctly.

Together with the scarcity of memory, this interrupt-based handling of communication events will dominantly influence the way communication facilities for embedded systems are designed.

CSP-based formal checking is easy for any piece of code built using CSP-based channels, processes and constructs as building blocks. After these basic building blocks are made, one is tempted to build all higher level

layers of the CSP-based library including the communication subsystem using this application model. Pairs of `transmit` and `receive` processes for each physical link on the embedded-system board can be made to prioritize and synchronize access to communication resource in each direction. Those processes would have to be active all the time after the initialization phase. Inside an ISR that handles arrival of packages over a link, received packages are directed to some intermediate buffer and this event is signaled to `receiver` process controlling input from that link. But, the maximal number of packages, waiting to be handled by the receiver process and directed further to the consumers, cannot be known in advance. Thus, dynamic memory allocation must be used for making new elements to be added to the data buffer. Unknown size of those buffers prevents application of this approach in memory scarce and real-time constrained embedded systems.

If, however, there are no central processes and associated central buffers, memory buffers can be associated with channels. Since each channel can handle only one message at the time, the buffer size needed in each channel is known in advance and memory can be pre-allocated. Necessary condition is that types of data carried over channel are statically defined prior to run-time. Thus, dynamic creation of channels and changing data types sent over existing channel is not allowed in strictly embedded implementations.

Scarce memory constraints combined with time unpredictability of memory allocation leads to implementing offline static schemes. Nevertheless, even in a system where all memory is pre-allocated in advance, dynamic system reconfiguration is possible. If processes and channels needed for any mode do already exist, the mode-switching event will just trigger using the *alternative* construct/process hierarchy. However new modes cannot be added dynamically, nor new processes can be downloaded.

A. *Encapsulate link protocol functionality in* `link driver` *and device specific details in* `device drivers`

In this paper Hilderink's `link driver` concept is extended. While in Hilderink's concept, a good decision was taken to decouple program structure from the way of distributing, practice had shown that handling reliability, heterogeneity, portability and reconfigurability demands further decoupling.

If control loops with stringent real-time requirements cannot be implemented in a single processor, communication is usually performed over fieldbuses. In fieldbuses, maximum message size per single bus transfer is limited. Large messages are divided in several packages. Specific *communication protocols* are defined to ensure proper assembly/disassembly of packages/messages on both sides. In the communication protocol between two channels fieldbus specific properties like packet structure and maximal size, should be decoupled from channel specific properties like data representation and security policy issues.

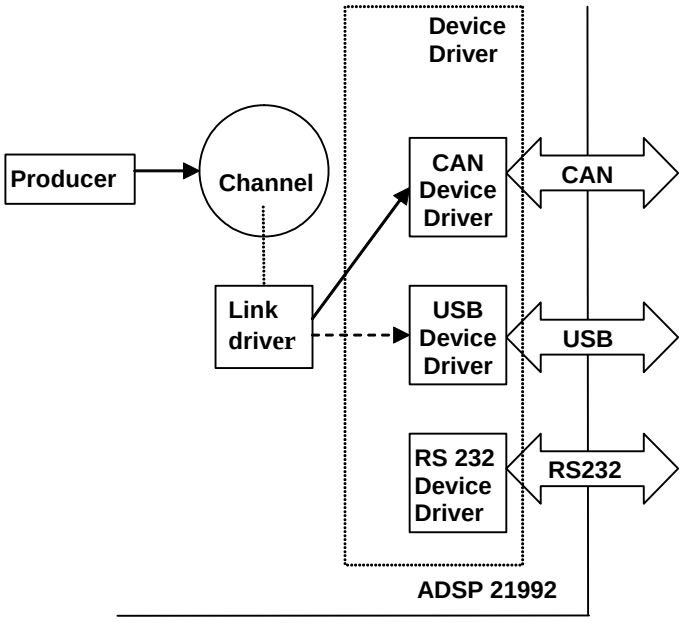

Figure 2: Modified concept of implementing link and device drivers

The fieldbus and device dependent part of the communication protocol is encapsulated in the form of the *package protocol* and the part associated with logical channels is encapsulated in the *link protocol*. This protocol decoupling reflects itself in the division of Hilderink's link driver into two parts: `link driver` and `network device driver`. Notion of `device driver` is added to handle all device specific details including those not concerned with communication. Network device drivers are in functional sense parts of device driver. As in the basic link driver concept, all driver objects are passive. There is only one instance of a device driver per board and one instance of a network device driver per each network interface of the board. On the other hand each remote channel has one *link driver*. This modified concept is depicted on Figure 2.

Functionality and register configuration specific to the target hardware is encapsulated in device driver objects. In each device, driver support is made for communication over various links that the board contains. All device details related to particular communication link are encapsulated in the appropriate *network device driver* (`CANDeviceDriver,`

`USBDeviceDriver` and `RS232DeviceDriver` in *Figure* 2). Functions of those drivers set all registers needed to initialize fieldbus links, implement transmitting and receiving data and acknowledgment messages. Receive and transmit `ISR` logically also belong to those objects.

Data1
Data2
Data3
Data4
Node ID of transmitter
Channel ID of transmitter
1

The 1st package

Data1
Data2
Data3
Data4
Data5
Data6
n

The other packages (n is index number)

Figure 3 Simple package protocol defines structure of the CAN messages. Link protocol is hidden in data fields

Data is sent over network in packages made according to package protocol used in appropriate network device driver. Based on the size of transmitted message, needed number of packages depends on the applied network device driver. If packages could be lost, each package should encompass a number identifying the location of sent data fields inside overall message. In Figure 3, number n is such an index identifying location of field Data1 in the overall message. Although sending package number appears to be more simple solution, it would make link driver, who assembly data into the message, dependent of used package protocol. Such a solution would significantly complicate switching to the alternative physical links in case that the primary link fails. In case of dynamical channels, the first package should also carry the information concerning a total message size. In the *rendezvous* communication an acknowledgement (ACK) signal needs to be sent back across the fieldbus to release the producer after all data has been received by the consumer. The address of the producer part of the remote channel is normally determined in advance in the link driver of the receiver prior to the compilation phase. If additionally, the channel can receive data from one or several producers or from a producer unknown at the compilation time, the package protocol is made such that the first package carries the address of the producer side of the remote channel. In this case, receipt of the first package should be acknowledged as well. Simple CAN package protocol can be defined as on Figure 3.

Other device-specific details concerning peripherals (A/D, D/A, timers, digital inputs/outputs, DMA, etc.) are encapsulated in the `deviceDriver` object. These device drivers also take over the role of *network and transport OSI layers* by taking care of routing if it is necessary and handling exceptions caused by failure of network links. In case of physical link *failure*, a so-called *device driver* will provide link drivers with an alternative network device driver if possible.

The `link driver` is now totally independent of device details. It is unaware of physical links and routing used and the way physical communication is implemented. Package protocols are fieldbus specific and therefore unknown to functions of the LinkDriver object. Network device drivers must therefore extract from packets data fields that are of interest to link drivers. The link driver is concerned only with the link protocol used, or more precisely it takes care solely of the session reliability, data representation and security policy (roles of session and presentation OSI layers). Link driver takes care that there is no data loss or duplication. Only link drivers speaking the same language (having the same link protocol) can understand each other and communicate. Each pair of communicating link drivers can use a custom protocol. This can be used for instance to encrypt information using a custom security policy known only to both sides of that channel. The rest of system is unaware of the used link protocol details. Link protocol changes are done simply by substituting link drivers, leaving the program structure and functionality untouched. Two link drivers using the same link protocol can communicate to each other over various links. Furthermore, using `Request_Retransmission` functionalities of network device drivers, link drivers can safely complete started session even over alternative physical links.

Error detection and recovery can be encapsulated in either one or both protocols. For instance, due to its deterministic, reliable behavior (error detection based on CRC, bit stuffing...), `CANDeviceDriver` object will not need any additional error detection and recovery mechanism. `RS232DeviceDriver` might need one since underlying serial bus uses only optional parity checking scheme. If part of data is found to be corrupted and recovery is not possible, retransmission is requested.

Custom link protocol details and device details are decoupled. One is tempted to decouple general fieldbus protocol details from its device specific implementation. However, practice has shown that the amount of clear general fieldbus protocol code independent of device details is negligible. Nevertheless, softer decoupling can be used by applying ‘template method’ design pattern [9]. Template Method lets subclasses redefine certain steps of an generic algorithm without changing the algorithm's structure. General fieldbus specific behavior

defines the skeleton of an algorithm in an operation, deferring board specific steps to subclasses.

III. IMPLEMENTATION OF CAN DEVICE DRIVER FOR ADSP 21992 BOARDS

A. CAN specific functionality

The CAN protocol supports a maximum package size of 8 bytes and all larger structures must be divided into packages. The `MessageID` in the CAN 2B protocol has 29 bits and is used both for arbitration on the bus and content based addressing. Every node will filter `MessageID` of the incoming message and nodes interesting in that type of message will receive it.

In CAN protocol, message that wins *bitwise arbitration* will be the first to be transmitted over the bus. Arbitration is performed by comparing messageIDs bit by bit, starting from highest bit). Zero bit level is dominant. Active levels and order of bit fields in messageID must therefore be carefully chosen to result in proper priority of messages. The first part will influence dominantly priority of whole message and is therefore assigned to priority number obtained by combining the importance priority (needed to achieve graceful degradation in case of network congestions) and deadline priority (to enforce real-time message delivery) [10]. The second part of the message encompasses fields needed to distinguish address and type of message during filtering. Each side of a remote channel receives the local address on its node. The whole address of such part of the channel is consisting of number uniquely identifying node (`nodeID`) and number uniquely identifying channel on that node (`channelID`). Address should influence message priority in the least possible extent and is therefore put on the least significant bit fields of the messageID. Additional bits are added in the middle of the messageID to distinguish between messages and acknowledgments (request for retransmission is considered to be negative acknowledgment) and between synchronized and broadcast messages. Also to optimize number of packages sent additional bits can be added. Optimization is done by gluing control flow information (like End of Transmission (EOT), End of message (EOM) and their acknowledgments) in messageID of data packages.

For sending an acknowledgment, the source address (`nodeID + channelID`) of the producer side of the channel is needed on the receiver side. This address should however *not* be sent in messageID. Reason is that the number of bits available for the fields specified above would be too much reduced. In this way, it can be offline hard coded in channels for optimization purposes or it can be specified in a package protocol, like in simple CAN package protocol shown on Figure 3.

Each different implementation of a CAN fieldbus protocol has its own specific parameters and its own way of using. All those details are hidden in the board specific CAN driver inherited from more general `CANDeviceDriver` class.

B. Experimental setup

In this project, two ADSP-21992 EZ-KIT LITE boards were used. The ADSP-21992 is 16 bits fixed-point DSP processor from Analog Devices. The ADSP-21992 DSP is equipped with a DMA (direct memory access) controller and several on-chip peripherals: AD Converters, DA Converters, PWM output, timers, SPI port, flag I/O and a CAN interface. The ADSP-21992 comes with a development kit board called ADSP-21992 EZ-KIT LITE. On this board there are connectors for CAN bus to daisy chain the boards. Configuring the CAN controller means to set up the CAN speed and set values for filter masks of the appropriate mailboxes. This is the main task in configuration mode. The maximum speed of CAN bus is 1Mbps, unfortunately, Analog Device recommends using lower speed.

ADSP boards can use up to 16 mailboxes. Each mailbox can be configured either as transmit or receive. In our implementation, separate mailboxes are reserved to transmit and receive acknowledgments. Besides those two acknowledgment mailboxes, one mailbox is dedicated to transmiting data and all the other are configured to receive data. `CANTransmit()` and `CANTransmitAck()` functions of the `CANDeviceDriver` object hide the way of transmitting data and acknowledgment messages from their producers. Synchronization and prioritization of sender processes is hidden in the call to this function, such that after successful transmission only one process can pass the semaphore and access the CAN transmit mailbox. This hardware-specific object also offers functions used for CAN initialization (speed, mailbox configuration, interrupts initialization…).

In CAN, a hardware ACK is sent only when at least one of the nodes receives a package, otherwise the package will be retransmitted. It seems that there is no possibility to lose packages. If a package can not be lost, there is no need to send the index number in each package. However, experiments done with the ADSP boards showed that packages can be lost inside of the CAN receiver. If several mailboxes have the same ID, the message will go to the mailbox with the highest number. The other mailboxes will remain the same. This facility is used to make a back up for the receiver mailbox. When a message arrives at a certain mailbox and unfortunately the data have not been read yet and

that mailbox is in the overwritten protection mode, the back up mailbox is used. But if all mailboxes with the matching filter are full, data is lost in the receiver although the producer has got a hardware acknowledgment that the message is transmitted successfully. Therefore, each package must have a index number associated, as in package protocol from Figure 3. In protocol shown on that figure careful observer will notice that packets are not using maximum allowed size of 8 bytes. Due to the fact that smallest addressable data on ADSP boards is word (16 bytes), breaking one pair of data bytes in two subsequent packets is avoided. An index number is therefore occupying last fields in a package. In case of small messages, one byte will suffice for sending index and package will have size of 7 bytes. However, data packages with indexes higher then 255, will send 8th byte as well. Of course, this is just one of many possible package protocols and decision to change it will not spread out of network device driver boundaries.

Used IDE is equipped only with C compiler, which led to the development of communication framework for C version of CT library. C++ like object-oriented features were mimicked using several design patterns developed by Hilderink for the need of this library.

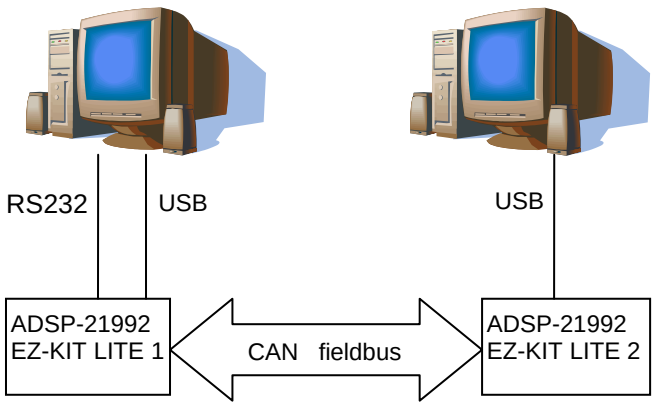

Figure 4 Hardware architecture model used in experimental setup

C. Detailed description of implementation

Synchronization and prioritization in accessing a network link can be handled even more efficiently and more naturally without the use of centralized receive and transmit processes. Producer and consumer processes involved in remote communication already have priorities assigned. A link driver is a passive object whose `write()/read()` function is executed in context of a user process (producer or consumer) after a call to the `write()/read()` function of the associated channel. The user-defined producer writes data to the channel not knowing whether that channel is remote or not. If the channel is remote, the `write` function of the associated `link driver` is called. In link driver, data is viewed as a stream of bytes consisting of data bits and redundant bits added to implement link protocol. In order to deliver the data to its counterpart on the other side of physical link, the link driver will make call to `transmit()` function of the appropriate device driver object. In this function, data is taken from link driver in chunks equal to fieldbus specific packet size. Synchronization with other potential producers using the same physical link is hidden in the form of a semaphore. In case of CAN this semaphore will guard access to transmit mailbox. Since the waiting queue on this semaphore is prioritized, access to the transmit mailbox and order of transmitting will be naturally prioritized based on the producers' priorities. All network device drivers inherit from a common parent class in which the prototype of `transmit()` function is defined. This polymorphism is used to switch to alternative network device drivers in case of a link failure. After successful transmission, in automatically invoked transmit interrupt service routine (`Tx_ISR`), the semaphore guarding mailbox access is signaled to release the highest-priority producer from the waiting queue. In this way, access to the fieldbus is prioritized naturally using the priorities of the producers. Packages from different nodes are then competing on the bus to win CAN bitwise arbitration and access the bus. The queue associated with the semaphore guarding access to the transmit mailbox is sorted each time a new package enters. Once low priority message enters transmit mailbox, heavy CAN traffic can cause it to waits long before wining arbitration and actually accessing the bus. In the meantime higher priority message might enter waiting queue of semaphore guarding access to mailbox. This *priority inversion* problem can be resolved by canceling transmission and sending low priority message back to the waiting queue.

Link driver on the consumer side can use the possibility to request retransmission of lost data. For transmitting positive and negative (retransmission requests) acknowledgments similar implementation is used as for transmitting data messages.

Each remote channel in the system is uniquely identified based on its address. Such an address actually contains two parts: the address of the node (`nodeId`) and the local address of the channel on that node (`channelId`). The producer and consumer side of a channel thus have different addresses. The destination address is always sent along with the message and its source address is sent in case of many-to-one communication and in the case producer's identity is not known in the compilation time. Upon message arrival to the receive mailbox, a receive interrupt (`Rx_ISR`) is invoked automatically. In the context of `Rx_ISR`, CANDeviceDriver will extract the destination

`channelID` from messageID, and use it to search for link driver responsible for that channel. If the number of remote channels per node is not too much compared to the available memory (almost always true), this searching of the appropriate link driver can be optimized using a lookup table, implemented as an array of link driver pointers with `channelID` as index entry. To enable switching to alternative physical link in case originally assigned one fails, this lookup table should for each entry define also primary and alternative links. The Rx ISR will use functionality of device driver to extract data field important to link driver from packet. Link driver functions will be invoked to copy data to the already allocated buffer space. Note that the ISR should waste *no* time on memory allocation, because ISRs should consume as less time as possible. Calling any potentially blocking synchronization primitive (like rendezvous channel communication) is not permitted from inside a ISR. Simple synchronization primitives, provided by Hilderink in original link driver framework, are used to deschedule consumer process if there is no activity it can perform and to reschedule it when appropriate conditions are met. This one-sided synchronization can be done since protocol is made such that same parts of the buffer are not used at same time from contexts of ISR and Consumer process. This is possible because we are aware of the fact that Consumer process can not preempt ISR. Since there is no danger of corrupting data, mutual exclusion synchronization is not needed, resulting in elimination of possibility for blocking inside an ISR. After last package or glued with it, *End of Transmission* (EOT) message is sent. Reliable protocol requires acknowledging this message. If not acknowledged after timeout expires message is considered to be lost and it is sent again. Link driver of consumer will be rescheduled after receipt of EOT message. It will check received data and send positive acknowledgment or negative acknowledgment (request for retransmission of certain data fields). The address of the producer is either fixed before compile time or sent as part of the first package. On the node where the producer is, the acknowledgment will arrive into the receiveAck mailbox. The Rx ISR will be invoked, and after using `channelID` to identify the `linkdriver`, it will reschedule the producer process. In case of positive acknowledgment Producer will continue its execution. In case of negative acknowledgment because some data is missing or corrupted, minimal subset of data will be retransmitted. Using timeouts can be exploited to identify failed links or nodes.

D. Relaxing scarce memory constraints

Memory needed for buffers to gather and assembly packages on the receiver side is in embedded systems allocated in advance. If memory constraints are relaxed, dynamical channel creation and sending the protocol before communication takes place are also possible.

The protocol can be sent from the link driver of the producer side, each time the protocol is changed. This allows the possibility to send any data to a channel and to arbitrary change the type of the data sent. Because sizes of buffers are not known in advance, the memory space needed can not be pre-allocated in the initialization phase. As already explained, dynamically reconfigurable channels should not be used for time or memory critical embedded systems. Memory allocation of those buffers is not allowed in ISR, but if it is done during the protocol negotiation phase in the context of the Consumer, it will not jeopardize the real-time behavior of the system.

Another possibility is to make new channels dynamically and define the protocol when they are made or to reuse channel structures from unused modes. Protocol data is in this case sent during channel creation by a background process managing system reconfiguration. The protocol information is sent as any other data using the `Transmit()` function of the `device driver`. The main difference in case of the CAN implementation is in the protocol control/data bit of CAN `messageID`.

IV. CAN REAL-TIME CONTROL LOOP BE CLOSED OVER FIELDBUS?

Hard real-time control data is usually exchanged over serial fieldbus protocols. Following the improvements of the occam architecture [11], the current CT library framework offers a possibility to statically attach priorities to parallel processes using the PRIPAR construct. In combination with rate monotonic priority assignment, this will work fine as long as the ystem is made in such a way that no control loop is spanning over several nodes.

If control loops are not passing boundaries of nodes, the actual design will consist of several hierarchies of CSP constructs and processes that will run in parallel on different nodes is a satisfying programming model. However, such an application model, inherited from *occam*, in which only parallel construct can be distributed (PLACED PAR), is not target-architecture independent. An application model is independent of the target architecture if an application designer can make it exclusively based on the required functionality. According to [12], separating application and target hardware architecture models of embedded systems plays a key role in establishing a high degree of modelling and

exploration flexibility. Later in the design phase, the same application model can be mapped onto a range of hardware architectures.

Closing control loops over a network, demands a somewhat changed software application model. The application model decoupled from the architecture model implies that any process (keep in mind that constructs are processes too) should be deployable on any node. Thus, all constructs should be extended to be distributed in a similar way the PAR construct is extended with PLACED PAR in occam. Exception is the PRIPAR construct, which was derived to assign priorities in using a basic shared resource i.e. the 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 sense to use it in the usual way. From theoretical point of view, there seems to be no severe obstacles to make distributed versions of SEQ and ALT constructs. Different parts of sequential behavior can be executed on nodes where the necessary resources are. Such a sequential behavior is actually implemented by using parallel processes on different nodes. But semantically this behavior is sequential since the end of one process from a sequence triggers the next one and the first process in sequence can be triggered again only after the last process in the sequence finishes. The ALT construct requires additional communication overhead to synchronize guards attached to remote channels.

However current implementations of those constructs are not suitable for the new application model. Reason lies in the fact that currently in practical implementations of occam-like libraries [5, 6], constructs are not realized in the same way as other processes and instead of communicating only through channels, function calls are used. Furthermore, parent constructs are not protected from knowing the identity of their subprocesses. Note that 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. The essential reconfiguration power of CSP lies in the fact that processes do not know about the identity of processes they are communicating to. 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, a process and its parent construct need just to be correctly connected through a pair of channels. Thus, instead of using dedicated function calls, the start of a process execution is triggered over a channel and the end of process execution is signalled to its parent construct using another dedicated channel. The start channel can be used to pass mode change information to the process and the process can pass its status concerning its level of success in performing its service using the end channel.

Precedence constraints of tasks from each control loop can easily be translated into a hierarchy of CSP constructs and processes. However, control loops have stringent real-time requirements and this new application model does not give us a clue how to achieve real-time guarantees. Applying a Rate Monotonic (RM) approach is not possible any more, because RM priorities can be defined only on a global system level and scheduling is done on each node separately. Deadline based approaches, with conservative deadlines derived for every process execution and channel communication, seem to be the only feasible way to achieve real-time guarantees for dependent processes distributed over several nodes [10].

V. CONCLUSIONS

In this paper, modification of the distribution model implemented in our CT library is proposed. The existing link driver model is an advanced concept involving modular composition and applying object-oriented principles. However, implementation of CAN link drivers led to a more fine-grained modularization and separation of concerns in order to enhance reliability, flexibility and portability. The extended link driver model is implemented and implementation issues are described here. However, after concluding that the occam application model applied in our CT library is still not applicable in cases where a control loop spans over several nodes, modifications to this application model are suggested. Proposed modifications, must be more seriously analyzed before they are implemented.

REFERENCES

[1] A. W. Roscoe, *The Theory and Practice of Concurrency*: Prentice Hall, 1997.

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

[3] W. A. Burns A., *Real-Time Systems and Programming Languages Ada95, Real-time Java and Real-Time POSIX*, Third ed. Essex, UK: Pearson Education Limited, 2001.

[4] P. H. Welch, M. D. May, and P. W. Thompson, "Networks, Routers and Transputers: Function, Performance and Application" http://www.cs.ukc.ac.uk/pubs/1993/271, 1993.

[5] P. H. Welch, "Java Threads in the Light of occam/CSP," presented at Architectures, Languages and Patterns for Parallel and Distributed Applications, WoTUG-21, Amsterdam, 1998.

[6] G. H. Hilderink, J. F. Broenink, and A. W. P. Bakkers, "Communicating threads for Java,"

presented at Proc. 22nd World Occam and Transputer User Group Technical Meeting, Keele, UK, 1999.

[7] G. H. Hilderink, A. W. P. Bakkers, and J. F. Broenink, “A distributed Real-Time Java system based on CSP,” presented at Proc. Third IEEE Int. Symp. On Object Oriented Real-Time Distributed Computing ISORC'2000, Newport Beach, CA, USA, 2000.

[8] N. Schaller, G. H. Hilderink, and P. H. Welch, “Using Java for Parallel Computing JCSP versus CTJ, a comparison,” presented at Communicating Process Architectures 2000, WoTUG-23, Canterbury, United Kingdom, 2000.

[9] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*: Addison-Wesley, 1995.

[10] B. Orlic and J. F. Broenink, “Real-time and fault tolerance in distributed control software,” presented at Communicating Process Architectures 2003, Enschede, Netherlands, 2003.

[11] A. E. Lawrence, “Extending CSP,” presented at WoTUG-21: Architectures, Languages and Patterns for Parallel and Distributed Applications, 1998.

[12] P. v. d. W. A.D. Pimentel, E.F. Deprettere, L.O. Hertzberger, J. T. J. van Eijndhoven, S. Vassiliadis, “The Artemis Architecture Workbench,” presented at Progress workshop on Embedded Systems, Utrecht, the Netherlands, 2000.