Redesign of the C++ Communicating Threads Library for Embedded Control Systems

Bojan Orlic and Jan F. Broenink
Twente Embedded Systems Initiative,
Drebbel Institute for Mechatronics and Control Engineering,
Dept. of Electrical Engineering, University of Twente,
P.O.Box 217, NL-7500 AE Enschede, The Netherlands
Phone: +31 53 489 2817 Fax: +31 53 489 2223
E-mail: B.Orilc@utwente.nl

***Abstract*– The occam programming language, based on the CSP formal algebra and tied to a specific hardware platform (transputers), offered a structured way to organize concurrency. The CT library [1], developed at University of Twente, inspired by occam is a kernel library implementing occam primitives in modern programming languages and for general purpose microprocessors. In this paper, a practical implementation of the CT library is explained and some aspects are compared to similar occam-like library developed at the University of Kent. The design of the CT-library internals is then revisited and proposed changes are implemented in the C++ version of the library.**

Keywords– **Embedded Control Systems; CSP; occam; concurrent programming**

I. INTRODUCTION

A. CSP approach to modeling concurrency

From a laymen's point of view, software concurrency is always about some bunch of processes that can exist and execute simultaneously. But those processes rarely stand in isolation; they usually have to synchronize, exchange data and collaborate to accomplish their objectives. Concurrent software can be based on a synchronous communication model, an asynchronous communication model or some mixture of the two.

Software based on an asynchronous communication between components, models the world in terms of independent tasks accessing shared objects. Well established synchronization primitives (like events, semaphores, monitors) exist with a purpose to provide a mutually exclusive access to the shared objects. Moreover this kind of synchronization primitives is supported by all modern operating systems.

On the other side, most of formal theories assume that communication between components is synchronous. CSP is one of the first and still a popular formal algebra. CSP theory models the world as a set of collaborating processes that synchronize on events [2]. The first process ready to engage in some event must wait till every process participating in that event becomes ready to perform that event. Only after this event is performed, the processes involved are allowed to proceed. From obvious reasons, this type of communication is known as rendezvous communication.

Well established scheduling theory, that offers large variety of priority assignment schemes, is based on the asynchronous model [3]. These schemes do not work for the synchronous model. Therefore the asynchronous model fits best when focus is on time requirements. Rendezvous based design is on the other hand better suited when focus is on using formal methods and structured way of using concurrency.

The CSP theory had its simplified practical implementation in occam / transputer platform [4-6]. Occam is parallel language in which every statement is considered to be an elementary process. Blocks of statements are grouped into more complex processes using SEQ, PAR and ALT constructs. The *Sequential construct* (SEQ) defines strict order (or sequence) of execution for associated group of processes. The *Parallel construct* (PAR) defines that associated group of processes is executed concurrently. When a process offers to the environment a choice between several alternatives, those alternatives are grouped in an *Alternative construct* (ALT). Occam code can be transformed to machine readable CSP form and used in combination with existing formal checking tools, like FDR [7].

In occam, communication and synchronization is possible only through *rendezvous channels*. Call

*) This research is supported by PROGRESS, the embedded system research program of the Dutch organization for Scientific Research, NOW, the Dutch Ministry of Economic Affairs and the Technology Foundation STW.

channels and shared channels are the extensions defined by Occam 3 standard [8] which were never implemented in practice. *Shared* channels allow multiple processes to share one end of a channel. *Call channels* are needed in client/server paradigm where a request is send from a client to a server and result or a status of the performed action is communicated back to the client.

In occam, standard control flow structures like WHILE and FOR loops are used to implement repetitive processes. There are also two decision making constructs: IF and ALT. Differences are explained in more details in subsection II.B.

The occam / transputer platform was for a while really popular for implementation of complex control systems [9]. Control systems have strict time requirements, whose violation can bring controlled system in an unstable state. Real-time scheduling is therefore very important for control systems.

The CT libraries (CTJ, CTCPP and CTC) [1, 10], developed at University of Twente, as well as the CSP (JCSP, CCSP and C++CSP) libraries [11, 12] developed at University of Kent, are attempts to keep the benefits of occam programming language after its hardware life companion - transputers had disappeared. This is done by remarrying occam to existing hardware platforms and modern programming languages. Although occam was a simple language with simple semantic rules, not all of its aspects are trivial to efficiently implement in modern programming languages.

After the initial version of the CT library created by Hilderink was successfully applied in several control setups, the library has proved to be a good vehicle for further research in the area of distributed real-time control systems.

Software development of complex architectures is best viewed as an evolutionary process. In first versions, performance and structured design are often less important then proving the validity of the approach. The CT library is not exception to this rule.

In this paper, the design of the CT library is revisited in order to make certain aspects either more efficient or better structured. Focus is put first on the C++ version of library, because that version is most widely used in practical setups. Some of the performed evolution steps, are partly also a return to the origins of occam and transputers.

II. CT LIBRARIES

The CT Library is implemented in Java (CTJ), C (CTC) and C++ (CTCPP or CTC++). This library is customized for the application area of real-time control systems. The aim of the occam and CSP way is to supply users with safe patterns for organizing concurrency structure of an application. Although occam is based on synchronous rendezvous communication, the CT library also implements basic synchronization primitives created in models based on asynchronous communication, like semaphores and monitors. Those primitives are used as building blocks in the implementation of synchronous basic primitives (constructs and channel synchronization). It appeared as a more structured approach then building synchronization mechanisms from scratch each time those are needed. However, users of the library operate on a higher level of abstraction, where it is not allowed to use semaphores and monitors. The CT libraries, as well as the Kent CSP libraries, have additional features that distinguish them from occam.

A. Relation to object-oriented programming

While modeling the real world, humans rely on certain rules (abstraction, encapsulation, hierarchy, classification) in order to manage the complexity. Object-oriented programming (OOP) incorporates those rules as its basic principles. Concurrency of the real world is, however, better described using occam and CSP. In the time when CSP and occam appeared, the dominant programming style was imperative. Object-oriented programming, which is the preferred style nowadays, was still not conceived. Interesting research would be to fit those two concepts together.

Occam-like libraries are implemented using the OOP approach. But from the users point of view, OOP usage is in occam-like libraries restricted to lower level objects encapsulated inside processes. Maybe some other mixture better suited to human way of thinking can be discovered.

In occam, every statement is considered to be a process. Subprocesses can inspect variables defined in scope of their parent processes. But only if a parent process is sequential, subprocesses are allowed to modify those variables.

Though a process can be implemented as an object, essentially a process is *not* an object. While an object is an entity, a process is focused on defining some behavior, possibly a behavior attributed to an object, part of object or a set of objects. This behavior is specified by defining either strict orders of events (sequential execution) or by allowing events from parallel streams to be executed in any order, or by allowing a choice (alternative construct). Any partial behavior introduced in this way can be seen as a process. In the CT library, processes and constructs are implemented as objects, imposing more strict boundaries for variables scope then in occam. The scope of variables belonging to some process is, due to encapsulation, determined by the borders of the object implementing that process.

In occam, a logical condition associated with a guard, is an expression that can involve any of the variables

visible in the current scope and which produces as a result a Boolean expression stating whether channel communication is allowed or not. In occam-like libraries, the Alternative construct and Guards are also implemented as objects. Since encapsulation is a very important feature of the object, variables defined in the surrounding Alternative process are out of the scope of the guard object and can not be used in the expression to check whether communication on those channels is allowed. The CT library still leaves a possibility to do this by adding to each guard one Boolean variable representing the associated logical condition. Organizing expression checking and setting the Boolean representing the condition is left to the user process. In the Kent C++CSP library, it is chosen *not* to support such logical conditions.

B. Similarity and differences in occam, Kent libraries and CT libraries

In occam, the IF conditional statement is considered as a decision making construct, somewhat similar to the ALT construct. In occam, the IF construct is priority structured. The process associated with the first condition that equates to TRUE is executed. If no condition equates to TRUE, the resulting action is equivalent to STOP or deadlock. If it is possible that no condition equates to true, usually an additional branch is added with a condition that is always true and has as resulting action the SKIP process (process that does nothing, but successfully finishes). The branch to be executed is in the IF construct resolved directly when the IF construct is entered: there is no waiting involved. See the code below:

```
IF
   [logical condition 1]
      Process1
   [logical condition2]
      Process2
   [TRUE]
      SKIP
```

The ALT construct offers a choice between several events to its environment. Processes from this environment express their willingness to participate in particular events by attempting to communicate on the associated channel. Since those channels are guarded for communication attempts, the ALT will be notified about the chosen alternative.

If the environment in the same time accepts two or more events, this is resolved by choosing one of ready channels randomly (in theory), choosing the first ready channel from the list (PRIALT) or choosing the channel accessed by highest priority process (preference alting in CT library [1]) . Unlike IF, the ALT process will wait until one of the channels, for which the associated Boolean condition is set to true, becomes ready, and it will execute the associated process.

```
ALT
   [logical condition 1] & channel1 ?  data
      Process1
   [logical condition 2] & channel2 ? data
      Process 2
   [logical condition 3] & SKIP
      output ! x
```

In occam, only inputs can be guarded, there are no output guards. Of course, one can have output communication inside an ALT, and the SKIP process can be used in a guard in place of a channel communication (see branch 3 in above example). But this is somewhat different from an output guard, because there is no guarantee that the peer process on the input side of the channel is ready in the moment the ALT chooses this branch.

The CT library does allow output guards. It implements only one decision making construct – the Alternative construct. Behavior of the IF construct can be implemented using if/then control blocks of the native language (i.e. C++).

A guard that behaves the same as the combination of a logical condition and a SKIP process as in branch 3 of the given occam sample code is in the CT library called SKIP guard. A SKIP guard has no channel associated. Readiness of this guard depends only on the state of the associated logical condition. A guard that has no logical condition either, is in the CT library called an ‘else’ guard. An ’Else’ guard is always ready. If no other guard is ready, the alternative containing the ‘else’ guard executes the process associated with this guard without blocking. This is somewhat similar to non-waiting behavior of the IF construct. But worth noting is that Alternative is like a more advanced IF construct. If only SKIP guards and ‘else’ guard are used in Alternative, it will have the same behavior as an IF construct. Possibility to use readiness of the channels in guards makes the Alternative a superior choice-making construct.

Guards in the C++CSP library do not have a logical condition associated and thus there is no need to differentiate between ‘else’ and SKIP guards.

The prioritized version of the ALT construct (PRIALT) enables a more deterministic choice if more then two alternatives are ready for execution at the same time. Preference is given according to the order in which guards are specified in Alternative construct.

The C++CSP library has a facility that the same ALT construct can make a choice in different ways in subsequent triggering. Besides the expected fair ALT, and the PRIALT choice, it is also possible to give preference to the last chosen guard. This can be

beneficial to handle burst transfers without changing device settings.

The CT library also offers several innovations in a way a choice is made. The 'Preference Alting' mechanism enables that in case several guards are ready a choice is made according to priorities of the processes blocked on the guarded channels. However since different behaviors of the Alternative construct are captured using different classes; once the ALT or PRIALT object is defined, its behavior cannot be changed from one iteration to another.

The Alternative can in the CT library be executed in two ways: as a standalone entity possibly with its own context or as a selection-making statement in a context of a calling process.

The Kent C++CSP library prefers the second approach.

In our CT library, the Alternative is a kind of Guard. This might at first seem strange, but this is necessary in order to allow nesting of Alternatives – one of the features defined in the occam 3 reference manual.

C++CSP can have guards for buffered channels. Its associated guard is not ready during the time in which its buffer is empty. In the CT library, there is no guard for buffered channels. C++CSP defines several different kinds of guards: Normal Input Guard, Normal External Input Guard, Skip Guard, Buffered Input Guard, Relative Timeout Guard, Timeout Guard. All those different guards are implemented using hierarchy of classes with a common parent class – Guard. The CT library on the other hand puts all different functionalities of a guard in one object. Since Alternative is also kind of guard, it also implements all those features.

Both the Kent and CT libraries use the template mechanism of C++ as a way to make generic versions of channels that can be instantiated as a channel of a particular data type by specifying an appropriate data type as a template parameter.

One of the key aspects of the CT library is orientation towards the real-time systems. Therefore, a special scheduler is designed that uses the hierarchical structure of constructs and processes in efficient way and allows for a practically infinite number of priorities in a system.

C. First steps in redesigning CT library

First in the list of evolutionary steps for the CT library was to implement the context switch using the *fibers* mechanism of Windows. This might *not* seem a good idea for a library targeting towards real-time systems, but this step allows the use of best IDE tools available for further developments.

The C++ version of the CT library has a core of its kernel written in C. The second step was to transform all parts of the library to C++. This might seem as unnecessary step. It might look even as a step in a wrong direction since with this step maintenance of C and C++ versions is totally separated. But, if we take in account that C version of CT library has an object-oriented C++ look and feel, reasons for this change become more obvious. Also some glue code that connects C and C++ parts has been eliminated. But the main advantage is that, once whole library was rewritten in C++, it was far more readable and understandable.

III. Redesigning the Channel concept of CT library

A. Shared Channels

While the version of occam that was actually used was based solely on one-to-one rendezvous channels, occam 3, although never implemented as such, defines *shared channels* as well. According to that definition, a process that wants to use a shared end of the channel must first claim it, and then wait until its claim is granted by the process existing on the *not*-shared end of the channel. Basically this can be an 'any-1' or '1-any' type of channel, where the shared end is marked with 'any'. In practice, sometimes 'any-any' channels might be needed as well.

Communication performed on 'any-any' channels can be decomposed into 'any-1', '1-1' and '1-any' parts (see Figure 1). There are two shared or 'any' sides of the channel and on each of them a choice is made between processes attempting to access the channel. Pairs of processes that have passed shared sides of a channel can engage in a '1-1' rendezvous communication.

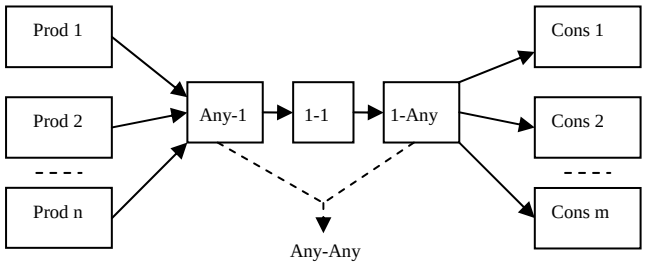

Figure 1: Any-to-any channel

In the Kent library, a distinction is made between different types of channels ('1-1', '1-any', 'any-1', and 'any-any'). The CT library on the other hand, implements only one type of channel that can be used as '1-1', '1-any', 'any-1' or 'any-any'. Since obviously, a channel used as '1-1' needs less synchronization then one used as an 'any-any' channel, it seems that performance might be better if channel types are separated. But actually, there are much more important reasons for separating channel types.

B. Channels in CT library

A channel is in the CT library implemented in a following way: on the shared ends of a channel, processes are sequentialized on two semaphores (readersSemaphore and writersSemaphore). A pair of processes that pass those semaphores will enter a Monitor, where synchronization and data exchange (communication) is performed. The Monitor contains three semaphores: *mutex* – that ensures mutually exclusive access to the monitor, *condition* – on which in this case the process that arrives first will be blocked waiting for rendezvous, and the semaphore *urgent* which is a standard part of monitor, used to block either a process released by signaling the condition semaphore or the process that has signaled the condition semaphore ensuring that only one process is active inside the monitor at any time. In this case, the process that arrives secondly performs the data exchange and afterwards releases the waiting process.

Besides the obvious overhead of this implementation, it is interesting that the synchronization primitives from asynchronous systems like semaphores and monitors are used as basic building blocks to implement rendezvous channels. In CSP theory, an opposite approach is used: synchronous communication is a basic building block and an asynchronous communication between producer and consumer process is implemented by inserting in between a buffer process that synchronizes on rendezvous channel with both producer and consumer.

Strangely enough, only the implementation of buffered channels in the CT library does not rely on semaphores and/or monitors which are naturally suited for this. Instead, probably in an attempt to achieve truly asynchronous behavior, the buffered channel throws an exception in cases when read is attempted from an empty buffer, and/or write is attempted to a full buffer.

C. Why are different types of shared channels needed?

A well-known rule in practical use of the CSP theory is that a channel cannot be guarded on both sides. Or in other words: two processes that engage in rendezvous communication cannot both be subprocesses of ALT constructs. The guarded side becomes ready to be chosen for execution only after the other side is already ready for execution. If both sides are guarded, then the associated processes will *never* become ready to be chosen. Since no process can proceed, this is deadlock.

An analogy can be drawn between shared ends of channels and the alternative construct, based on the fact that in both cases a choice is made between processes. Moreover one could implement a structure that will behave as an any-any channel using rendezvous channels and alternative processes. Of course, such an implementation would have significant overhead since several new processes are defined. But this approach is more for sake of description then for implementation.

The process that accesses the shared end of a channel is in a similar situation to a process that accesses a channel guarded by some ALT construct. In other words, a shared end of a channel is implicitly guarded. If a process accessing a shared end of a channel is a subprocess of some ALT construct, that channel is practically guarded from both sides.

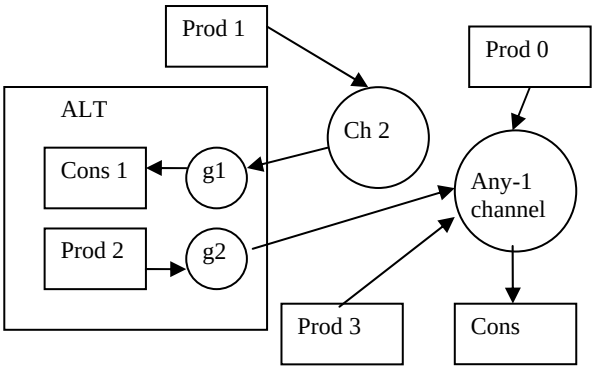

Figure 2: Example of an alt and an any-to-any channel.

In the example shown in Figure 2, g2 is output guard which guards the output from Prod2 to the Any-1 channel. Two implementations and two race hazard scenarios, equally unwanted, are possible:

In first scenario, the guard *is* allowed to claim the shared end of channel. If the ALT is triggered before Prod0 and Prod3, then g2 claims the channel and becomes ready. If in meantime g1 has also become ready Cons1 might be chosen for execution instead of Prod2. Prod0 and Prod3 cannot access the channel because it was already granted to g2.

If a guard is *not* allowed to claim a channel, then after process Cons has accessed the channel, guard g2 will become ready, but the channel might be granted to one of the processes Prod0 and Prod3.

Thus, the safe solution is to distinguish between ‘1-1’, ‘1-any’, ‘any-1’ and ‘any-any’ channels and to forbid output guards associated with ‘any-1’ channels, input guards associated with ‘1-any’ channels, and any kind of guards associated with ‘any-any’ channels.

Since like in occam, only input guards are allowed in the Kent libraries, an exception is thrown on attempt to associate input guards with ‘1-any’ and ‘any-any’ channels. The Kent library implements ‘1-1’ channels in a very efficient way, much alike to the mechanism that was used on occam/transputer platforms.

D. Redesigning channels

Rendezvous channels

Following the discussion presented in the previous subsection, the first decision was to separate ‘1-1’, ‘1-

any', 'any-1' and 'any-any' types of channels like it is done in the Kent library. After this change is implemented, the CT library becomes capable to detect and prevent misuse of guards on shared ends of channels. Similarly to the Kent library, the '1-1' channel is then implemented in an efficient way alike to the way used by occam / transputer platform.

In transputers [6] only one memory word was used to implement rendezvous channels. The process that arrives first, writes its Process ID (which is address of its workspace in memory) in this memory location. The process that arrives second performs data exchange and reschedules the first process. This mechanism was rather easy to mimic in the implementation of One2One channels for the CT library: Instead of using synchronization mechanisms like semaphores and/or monitors, one pointer to a thread (waitingProcess) was sufficient. First process that arrives needs to put pointer to its thread in waitingProcess variable and to deschedule itself. Those two activities should be executed in an atomic fashion. The process that arrives second performs the data transfer and reschedules the first process. In CT library this looks like the following code:

```
void One2OneChannel::write(Object* object,
unsigned size) {
   Processor::enterAtomic();
   if(!bReaderReady) {
      // wait for the consumer
         waitingProcess
               =Dispatcher::getRunning();
      bWriterReady=true;
      buffer=object;
   }
   else{
      Processor::copy(object,buffer,size);
      Dispatcher::add(waitingProcess);
      bWriterReady = false;
      bReaderReady = false;
   }
   //potential context switch
   Processor::exitAtomic();
}
```

The Read() function is designed in analogue way.

Shared channels

Relationships between the mentioned types of channels, naturally define a class hierarchy with '1-1' channel as a base class. The 'Any-1' and '1-any' channels extend the '1-1' channel by adding mechanism to arbitrate channel access on shared input and output end respectively. The 'Any-any' channel naturally inherit from both 'any-1' and '1-any' channels because it needs the access arbitration mechanism both on input and output end. The same class hierarchy also exist in the Kent C++CSP library.

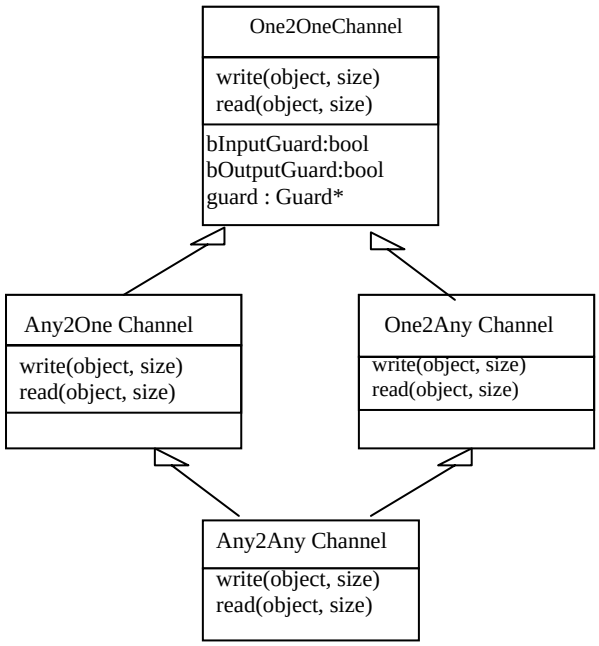

Figure 3: Class hierarchy of channels

The Any2One channels add the access mechanism on shared input end of a channel. A process which passes arbitration can engage in rendezvous with the consumer process using the functionality defined in '1-1' channel. The simplest way to implement this is to use an additional semaphore (writersSemaphore) on which producers are sequentialized before entering the inherited One2One rendezvous channel.

```
void Any2OneChannel::write(Object* object,
unsigned int size){
   writeSem->p();
   One2One::write(object,size);
   writeSem->v();
}
```

An analogue mechanism can be used for the One2Any channels.

```
void One2AnyChannel::read(Object* object,
unsigned int size)
{
   readSem->p();
   One2One::read(object,size);
   readSem->v();
}
```

The Any2Any channel can then inherit functionality of both the Any2One and One2Any channels.

However, there is serious disadvantage associated with the described approach.

In the CT library, Semaphores are implemented in such a way that processes in waiting queues are ordered by priorities. Still, the described access mechanism does not ensure that highest priority reader and highest priority writer will actually win the arbitration. Let us for

example imagine the situation in which several producers subsequently attempt to claim the channel and then after a while a consumer arrives. The first producer that has arrived is the first one that passes the semaphore regardless of its priority and only subsequent producers are ordered by priority in a waiting queue.

Instead of semaphores, a better access mechanism can be implemented by handling directly waiting queues (writers and readers) to which all incoming processes are added. Only after detection that the other side is ready, arbitration will be performed and a pair of the highest priority processes from the waiting queues will be released to engage in rendezvous communication. As an example of this approach, read and write function of Any2One channel are shown. The Reader accesses the not- shared end of the channel. Therefore it can directly go to 1-1 part. If there are writers blocked on the access mechanism of the shared end (in writers queue), reader will release the first one before entering the 1-1 part. If, however, reader enters the channel first then it will set the readerReady flag and it will block inside the One2OneChannel::read function. The first writer that arrives will proceed to the One2OneChannel::write function.

```
void Any2OneChannel::read(Object* object,
unsigned int size){
   Processor::enterAtomic();
   if(writers->getSize()>0 ){
      bBusy = true;
      Dispatcher::put(writers->get());
   }
   Processor::exitAtomic();
   One2OneChannel::read(object, size);
   bBusy = false;
}

void Any2OneChannel::write(Object* object,
unsigned int size){
   Processor::enterAtomic();
   writers->put(Dispatcher::getRunning());
   if(bReaderReady && !bBusy)
        Dispatcher:: put(writers->get());
   Processor::exitAtomic();
   One2OneChannel::write(object,size);
}
```

The write and read functions of One2Any channels are implemented in an analogue way. Any2Any channels are slightly more complicated because the access mechanisms exist on both ends of the channels.

Buffered channels

Instead of throwing exceptions, a buffered channel is now implemented in the standard way using synchronisation based on three semaphores with following symbolic names mutex, dataAvailable and spaceAvailable.

The equivalent occam and therefore also equivalent CSP representation can be easily constructed for this kind of bounded buffer, as shown below:

```
Buffer[size];
int spaceAvailable=size; int dataAvailable=0;

 WHILE(TRUE){
   ALT{
      [spaceAvailable] in? buffer[inputIndex]
         dataAvailable++;
         spaceavailable = size-dataAvailable;
      [dataAvailable]out!  buffer[outputIndex]
         dataAvailable--;
         spaceavailable = size-dataAvailable;
   }
}
```

The original implementation of a bounded buffer, although giving illusion of being truly asynchronous, could not be represented with an equivalent CSP block and would rely on channel users to deal with exceptional conditions probably by adding blocking synchronization.

IV. ALTERNATIVE AND GUARDS

In the previous part of the paper, a notion of the alternative construct and a way it can be used were described. Let's now focus on implementation aspects.

A. Transputer implementation

On transputer systems support for the Alternative construct of occam was built in on microcode level. Its simplified scenario was as follows: First the alternative start instruction is executed, which will bring the alternative process into the ENABLING state. In this state ALT executes the instruction enable channel for each guarded channel; if the peer process is ready and waiting on channel, then the associated guard becomes ready and the state of Alternative process will become READY. The Alternative wait instruction is then executed. If there are no ready guards, the state of Alternative process will be changed from ENABLING to WAITING and Alternative will be descheduled. When a peer process arrives to one of the guarded channels, the state of Alternative process will be changed from ENABLING or WAITING to READY. This will reschedule the alternative process. When a processor is granted to the Alternative process, it will execute the disable channel instruction for each of guarded channels. During guards disabling, choice among ready guards is made. In the instruction alternative end a jump is made to the code of the process guarded by the selected guard.

While the Kent C++CSP library implements exactly the same mechanism as the one in the transputers, the CT library has somewhat different implementation.

B. CT library implementation

Although the mechanism implemented in the CT library seems to work fine, responsibilities of the involved objects (channel, guard and alternative) are not clearly delegated, making verification, maintenance and readability of the code somewhat cumbersome.

The synchronization mechanism used inside is again a Monitor, which is again overkill. The Alternative is triggered by calling the select() function which first acquires the monitor. This function will return a selected guard to a caller. If no guard is ready it will block on the condition semaphore of the monitor. If some process attempts to access a guarded channel, its associated guard will be put in an alting queue and the condition semaphore will be signaled. AltingQueue is a helper class which will sort ready guards according to the 'preference alting' mechanism.

So far, on this global level design seems good. But let us now plunge slightly deeper in design.

Although there is a class that implements guards, it is not designed elegantly. Actually, the whole mechanism could be implemented in a more clear way, perhaps by using the transputer implementation as a guidance. Furthermore, a channel does not know whether its associated guard is input or output. Although this would normally be resolved after attaching a guard, in the CT library this is determined every time guards are enabled and/or every time access to the channel is attempted. During the guard enabling phase, the presence of writers and readers blocked on the channel is tested. If there is a waiting writer then the guard must be an input guard. In analogue way, if there is at least one reader waiting this must be output guard. In both cases, the guard is added to an alting queue. If there is no process blocked on a channel, one can not determine at this moment whether the guard is an input or an output. This will be resolved at the moment the first process accesses the channel: if it is attempting to write then associated guard, if any, must have been input guard. Then, the input guard will be added to the alting queue. An analogue scheme takes place when the process is attempting to read.

The actual scenario is more complicated because a guard can also be in a SKIP or timeout mode and certain things need to be done in case the alternative is nested and used as a guard inside its parent Alternative. Timeout and SKIP should not be modes of a guard. A better structured and easier implementation would be to implement those functionalities as special kinds of guards. This is for instance done in the Kent C++CSP library.

Instead of the recurring checks whether a guard is input or output, a flag can be used. This flag can be automatically set at the moment the guard is created. If the input interface of a channel is passed to a guard constructor, then the guard knows that the input side of the channel is guarded. Once a guard notifies its associated channel about its presence and type, it does not have to be anymore aware of the channel or even of its own type. A channel has to be aware of its associated guard. The channel is the one who notifies the guard whether there is a process waiting on the unguarded side of the channel. Guard and Alternative should be aware of each other.

In PriAlternative, guards get priorities according to the order in which they are added to the construct. The PriAlternative class is derived from the Alternative class and uses the same alting queue and therefore the same sorting algorithm. While sorting the queue, priorities of processes blocked on guarded channels are compared first ('preference alting'). If those priorities are equal then priorities attached to the guards are compared. The PriAlternative of the CT library therefore does not have the same behavior as the PRIALT of occam, which compares only the priorities attached to the guards.

The possibility to use different strategies for guard sorting in subsequent triggering of the same alternative construct is not allowed in the CT library, although it was straightforward to implement and is available in the Kent libraries.

Obviously, there is room for improvement both in a sense of performance optimization and making design better structured, less complex and more readable.

The decision was made to redesign the whole mechanism, keeping one eye on a way it was implemented in transputers and other eye on existing CT library implementation.

C. CT Library redesigned implementation

The transputer implementation gives a good guidance how the Alternative construct can be efficiently implemented.

In the new implementation, a guard contains three Boolean variables: bAltingEnabled is true when Alternative is in ENABLE or WAITING state and false otherwise, bCondition keeps information whether a guard is enabled or disabled and bReady is the indicator whether communication was already attempted on the unguarded side of the channel. Instead of using local copies in an optimized version bAltingEnabled and bReady variables can always be fetched from Alternative and channel objects, respectively. In this explanation, local copying is used because expressions are somewhat easier to describe.

Like in the original CT library, ready guards are sorted using the AltingQueue. Instead of monitor synchronization one semaphore is used. More precisely, a binary semaphore known sometimes as event is derived from Semaphore. The obvious difference to the

Semaphore is that the maximal value of the internal counter is limited to one. A Reset() function is added that puts its counter back to zero.

Like in the transputer implementation, after triggering enableChannel() is performed for every guard. If no guard is ready, Alternative will wait on event guardReadyEvent. When some process attempt to access one of the guarded channels, its associated guard becomes ready, it is added to alting queue and guardReadyEvent is signaled. After the Alternative wakes up, it will call disableChannel() for every guard. This is again similar to the transputer implementation. The process associated with the guard that is first in the alting queue is selected for execution. See the code below:

```
unsigned Alternative::select(){
   queue->empty();
   guardReadyEvent->Reset();
    enableChannel();
   if(queue->getSize() == 0){
      Processor::enterAtomic();
      guardReadyEvent->p();
      Processor::exitAtomic();
   }

   disableChannel();
   selectedGuard = queue->show();
    return selectedGuard->getIndex();
}
```

Enabling channels is actually done by going through the list of guards and enabling channel for every one of them, as shown below:

```
void Alternative::enableChannel()
{
   Processor::enterAtomic();
   for(int i=0;i<guards->getSize();i++)
      guards[i]->enableChannel();
   Processor::exitAtomic();

}
```

The guard should remember that it is enabled. If the associated channel is already ready and if the guard is enabled, this guard can be put to the alting queue. The Alternative is notified about that using the SetReady() function:

```
void Guard::enableChannel(){
  bAltingEnabled=true;
  if(bCondition && bReady)
     myAlt->SetReady(this);
}
```

In the SetReady function, the guard is added to the alting queue and an event that guard is ready is signaled.

If the alternative construct is nested, it should act as a guard. Its functions enableChannel() and disableChannel() are called directly from its parent like is done for any other guard. After one of its guards is ready, the nested alternative should keep acting as a guard and should notify its parent that a guard is ready, but instead of pointer to itself, it will pass the pointer of its ready guard as an argument in the call to the SetReady() function of its parent alternative.

```
void Alternative::SetReady(Guard *guard){
 if(myAlt)myAlt->SetReady(guard);
 else{
      queue->add(guard);
      guardReadyEvent->v();
   }
}
```

If no guard is ready, the Alternative will block on the guardReady event. When a process attempts to access a guarded channel, prior to blocking a call to the SetReady() function of the guard will be made. The guard will now know that its channel is ready and, provided that the guard is enabled and alting is enabled, it can signal the Alternative. Otherwise it will notify the Alternative when proper conditions are met - either during the next enableChannel() call or at the moment the guard is enabled.

```
void Guard::SetReady(ProcessThread* ptBlocked)
{
   bReady=true;
   waitingProcessThread=ptBlocked;
   if(bCondition && bAltingEnabled)
      myAlt->SetReady(this);
}
void Guard::ResetReady(){bReady=false;}
```

In a channel's read() and write(), functions calls to Guard::SetReady() and Guard::ResetReady() are inserted to keep the state of the associated guard in accordance with the state of the channel.

```
void One2OneChannel::write(Object* object,
unsigned int size){
Processor::enterAtomic();
if(!bReaderReady){
       waitingProcessThread =
                Dispatcher::getRunning();
      if(bInputGuard)guard->SetReady();
      bWriterReady=true;
      buffer=object;
      Processor::exitAtomic();
   }
   else{
        Processor::copy(object,buffer,size);
        Dispatcher::add(waitingProcessThread);
        bReaderReady=false;
        bWriterReady=false;
        if(bOutputGuard)guard->ResetReady();
        Processor::exitAtomic();
   }
}
```

```
void One2OneChannel::read(Object* object,
          unsigned int size){
Processor::enterAtomic();
   if(!bWriterReady){
      waitingProcessThread =
               Dispatcher::getRunning();
      if(bOutputGuard)guard->SetReady();
      bReaderReady=true;
      buffer=object;
      Processor::exitAtomic();
   }
   else{
      Processor::copy(buffer,object,size);
      Dispatcher::add(waitingProcessThread);
      bReaderReady=false;
      bWriterReady=false;
      if(bInputGuard)guard->ResetReady();
      Processor::exitAtomic();
      }
}
void Guard::ResetReady(){bReady=false;}
```

After at least one guard becomes ready, the alting queue is not empty any more. Following the logic of the transputer / occam platform, all channels should be disabled. The Alting queue will be emptied next time the Alternative is triggered.

```
void Alternative::disableChannel(){
   for(int i=0;i<guards->getSize();i++)
      guards[i]->disableChannel();
}

void Guard::disableChannel(){
   bAltingEnabled=false;
}
```

Inspired by the Kent libraries, another significant change was made. PriAlternative does not exist anymore as a separate class. Instead Alternative can be in one of following regimes: PRIALT, PREFERENCE_ALT and FAIR . The Mode in which the Alternative works is now maintained in the AltingQueue which will sort guards according to its current mode. There is no need for a special PriAlternative class. Furthermore PRIALT is defined in the same way as in occam: guards are sorted solely according to their priorities.

A bounded buffer can now also have a guard associated. The Associated input guard is ready as long as its buffer is not empty. The associated output guard is ready as long as its buffer is not full. A guarded buffer should not be used in an Alternative that is working in PREFERENCE_ALT mode. Reason for this is that ‘preference alting’ selects a guard according to the priority for the process blocked on the *not*-guarded side of the ALT. Processes accessing a buffer are blocked only if read from an empty buffer or write to a full buffer is attempted.

V. SCHEDULING

A. Occam / transputer platform for real-time control systems

CSP theory is not aware of priorities and scheduling

A distributed system based on hierarchy of constructs and processes is scalable by nature. Reason is that a node can actually be interpreted as a complex process that uses associated physical links as channels to other processors. Scheduling is concerned with sharing the time of a processor or the bandwidth of a bus. There is no notion of the scheduling and priority assignments in the CSP theory. Parallel processes are considered to be truly parallel. Sharing processor time to execute several processes on one processor is an implementation issue. It will not influence the CSP model of the application. It will however dominantly influence its timing behavior. For real-time systems, satisfying time requirements is a key issue.

Scheduling in OS based on general purpose processors

Usually, scheduling is performed by the Operating System (OS) based on priority levels assigned to processes. In general purpose microprocessors, context switching between threads is performed in software. Since such a context switch incurs significant time overhead, the intended parallelism is more coarse grained. For instance, a context switch is avoided by executing an Interrupt Service Routine (ISR) on the stack of the currently executing thread instead of treating it as a separate process. This perhaps saves one context switch per interrupt, but it also complicates making context switch during ISR, which is needed in a case when a higher priority process is released due to this interrupt. A standard approach to scheduling is assigning priorities to processes either in static way (fixed priority schemes, e.g. Rate Monotonic (RM)) or in dynamic way (Earliest deadline first (EDF) and similar schemes).

Transputers and scheduling

In a case of the occam / transputer platform, scheduling is a joint activity of software (occam) and hardware (transputer).

Transputers were designed specially for message-passing parallel systems. Hardware within a transputer multiplexes virtual links between processes from different nodes to the one of the four physical point-point links (DS links). The bus that connects the memory, the CPU and the link peripherals was a crossbar 4x32 bits data address bus.

The transputer has a hardware scheduler that distinguishes two priority levels. The higher-priority level was most often assigned to processes that handle i/o communication. Those processes are triggered either by

external events (e.g. data arrival) or by timeouts on time channel calls. The functionality of those processes is equivalent to ISRs of general purpose processors. The main difference is that such processes have their own workspace (or stack in terms of general purpose processors). The overhead of an additional context switch per interrupt (called event in terminology of transputers) is not significant thanks to the fast hardware scheduler. For the same reason, parallelism can be more fine grained.

Occam language support for scheduling

In addition to the PAR (parallel) construct in occam a prioritized version of the parallel construct, the PRIPAR construct, exists. The priority in a PRIPAR construct is relative to the order of adding processes to the parallel construct. However on transputer platforms only two priority levels were supported and a PRIPAR was therefore used only on top level. Additional priority levels were often implemented in software.

Scheduling rendezvous-based systems

The chosen scheduling mechanism is not important for CSP formal checking. But structuring a program in the CSP way does influence schedulability analysis because it imposes rendezvous based communication. In rendezvous based systems, process priority does not influence dominantly the order of execution. The communication pattern of the overall system is determined by event based interaction of processes.

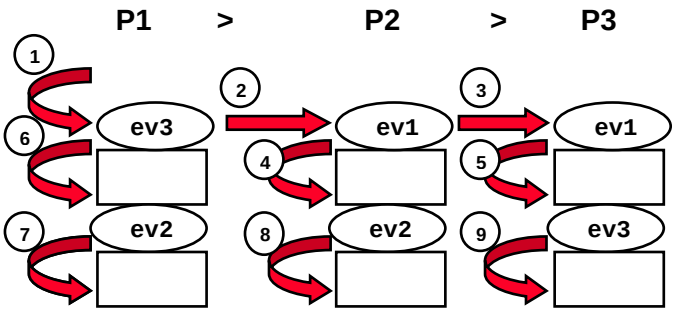

Figure 4: Process execution is dominantly determined by communication patterns rather then by process priority

Rules that govern such interaction are distributed in the communication patterns of participating processes. All those patterns together with process arrival times define only a partial ordering of the set of events. Still this partial ordering of events, inherently to the structure of the overall system, will always overrule process priorities. Assigning a higher priority to one process, engaged in a complex interaction scheme with processes of different priorities, does not necessarily mean it will be always executed before lower-priority processes. This is illustrated in Figure 4.

It seems that attaching priorities to processes does not impose enough influence on the behavior of rendezvous based systems. Maybe, it would be better to attach priorities to events. Deadlines are anyway time requirements imposed on events or on distances between some events. Furthermore, while priority of processes is local to a node, priority of events is valid for the whole distributed system. Only problem with an event based scheduling is that there should be a sound mathematical ground before it can be implemented in real-time systems.

Scheduling and formal checking in practice

Although in occam a channel communication between processes was rendezvous based, asynchronous communication was also used. It was implemented using an additional process that protected a buffer from simultaneous access by engaging in rendezvous synchronization with one or the other side (repetitive alternative on input and output channel).

If processes are communicating using asynchronous overwriting channels, established scheduling theories can be used to assign priorities. Actually, one can implement applications that contain a mixture of synchronous and asynchronous communication appropriate both for real-time scheduling and formal checking.

On the highest level, different priorities can be assigned to independent processes that exchange parameters through overwriting channels. E.g. in control applications those high level processes would be control and supervisory loops. The internals of those processes can in turn be implemented as a cooperation of processes of equal priorities communicating over rendezvous channels.

The top-level PRIPAR construct can order control loops by sampling frequency, which is equivalent to a RM (Rate |Monotonic) priority assignment. Communication inside loops is allowed to be rendezvous and all communication between loops is asynchronous. In [13] this idea is implemented in a way that it can efficiently use properties of a hardware scheduler.

B. The scheduling mechanism of the CT library

The CT library is in fact a kernel that, besides providing communication and synchronization primitives, also offers its own scheduling policy. This is very useful when a CT based program is executed on a bare microprocessor. A program based on the CT library can also be executed on the top of some operating system (OS). In that case, it will run in one system-level thread. All processes and stacks defined inside that thread are invisible to the OS. The scheduling policy of the CT

library will in this case manage the time that is granted to the associated system level thread by the OS. This concept is often referred to as a user-level scheduling. To achieve real-time guarantees it is necessary to run CT-based application as a highest priority process of real-time OS (RTOS).

There is an important problem associated with user level scheduling. Inside an OS, synchronization primitives like semaphores are often used. If one of the processes from the CT application attempts to make a system call, there is a chance that it will have to wait on a semaphore to access some resource. However the OS does not see CT processes, it sees only one system level thread (containing the whole CT program). Therefore, on an OS semaphore, the whole system level thread running the CT program will be blocked. Thus all CT processes inside that thread will halt after one of them issued access to some OS resource. This is not desired.

Preemption in the CT library

The CT library is not fully preemptive in a strict sense. A pair of functions (enterAtomic() and exitAtomic()) exists to mark the borders of an atomic section. enterAtomic() increments a counter of nested atomic sections and exitAtomic() decrements the counter and performs a context switch if needed, provided that the atomic section is not nested. Atomic sections are used in implementation of the CT library communication and synchronization elements-like: channels, semaphores and monitors. Only at those points in program execution, preemption can take place. This is in conjunction with the underlying CSP theory.

Hierarchy of dispatchers

Traditionally, the term *Scheduler* is used for a part that maintains a queue of ready processes (ready queue) and determines the process that will run next. *Dispatcher* is the term used for a part that actually performs the context switches from the running process to the next one determined by the scheduler. In the CT library, those functionalities are not distinguished and *Dispatcher* is the name used for a process that maintains a ready Queue. Actually, there is not one Dispatcher, but a whole hierarchy of them. Scheduling is based on a tree-like hierarchical structure of occam programs.

In the CT library, stack management is encapsulated in the ProcessThread class. Not every process needs to have its own stack. For instance, subprocesses of the Sequential construct can obviously all be executed by the same thread. Similarly, the process representing a chosen alternative can be executed by the same thread that has executed its parent ALT construct. Therefore, in the CT library, only for processes executed in Parallel and PriParallel a separate context (read stack) is needed and a separate ProcessThread needs to be created.

While all subprocesses of a Parallel construct have the same priority, priorities of processes inside a PriParallel are based on the order in which they are added to the construct. This is a fundamental difference between Parallel and PriParallel constructs.

Hierarchy of PriParallel constructs alone can allow an infinite number of different priority levels in a system. But a hierarchy containing both Parallel and PriParallel constructs, contains also some processes of the same priority. Most efficient way to organize a ready queue might be to divide it to several ready queues — one for each priority level. Since an infinitely large number of priorities can exist, a centralized dispatcher might contain too many ready queues. Instead, in the CT library a hierarchy of dispatchers is employed. For every PriParallel construct, one dispatcher is created. In this way, nested Parallel processes, which all have same priorities will go to one ready queue, belonging to the dispatcher of the closest parent PriParallel process.

According to the given explanation, if a top-level construct is *not* a PriParallel there will be no Dispatcher to manage the top level processes. In the CT library, when a Parallel construct is created and there is still no dispatcher in a system, then a dispatcher will be created for this top level Parallel construct. If a top level construct is an alternative or a sequential construct, in its constructor a hidden parallel construct will be created and made top level construct.

For a following hierarchy of processes, three dispatchers will be created:

```
PAR  P
   PRIPAR  P1
      P11
      P12
      P13
   PAR  P2
      PRIPAR  P21
         P211
         P212
         P213
      P22
      P23
```

One for the top-level Parallel construct (P), and one for each PriParallel construct (P1, P21). Processes P22 and P23 will be added to the top-level Dispatcher. The Process Threads of the two Dispatchers associated with PriParallel constructs P1 and P21 will be added to a top-level dispatcher. See Figure 5.

Process Threads for IdleTasks and Dispatchers

Every Dispatcher is executed in a separate Process Thread. When a parent dispatcher runs the Process Thread of a nested Dispatcher, this nested Dispatcher will take over and become current dispatcher of the processor.

As shown on Figure 5, a Dispatcher contains one prioritized Ready Queue. In addition, there is one FIFO Queue for every priority level of the dispatcher. The Process Threads of the nested Dispatchers will also appear in those FIFO queues. To efficiently manage ready queues in the associated dispatcher, a PriParallel is allowed to have up to 8 priorities. Of course any number of different priority levels can be achieved by nesting PriParallel constructs.

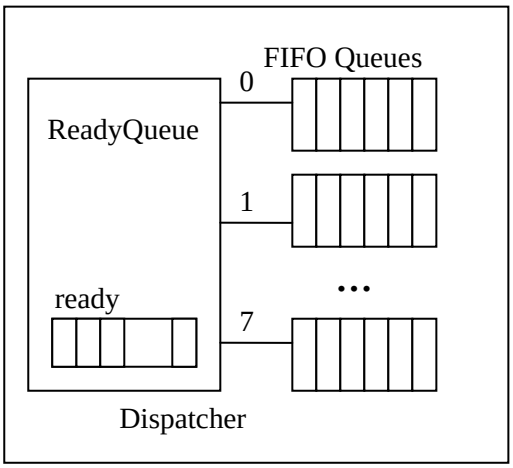

Figure 5: Dispatcher and ready queues of the example

Every Dispatcher contains an additional process thread known as IdleTask. This process thread will execute when all processes from the current dispatcher are either blocked or finished. The Idle task will give control back to the parent dispatcher. If the idle task of the top level dispatcher is executed, and not all processes have finished, and provided that the cause of blocking was not a deadlock, the program will have to wait on an external event to wake up one of the blocked processes.

In a case where a CT-based application is executed directly on a microprocessor, the top-level idle task performs waiting in a busy loop. When however a CT-based application is executed on a top of some OS, the CT program and all its processes run in one system level thread. To allow execution of other threads running on the top of the same OS, the Idle task will, on such platforms, suspend the thread in which the CT-based application is being executed.

C. Redesigning scheduling mechanism of the CT library

Starting point

Additional Process threads for every dispatcher and the associated IdleTask increase significantly the total number of context switches and associated overhead. Those processes are existing in every CT-based application, but hidden from the user. If a user decides to make a CSP model of its application, it is not likely that those additional processes will be included in such a model.

Let us try to get rid of these active processes and implement the functionality they offer in passive Dispatcher objects. The basic idea of a hierarchy of dispatchers remains the same. Like in the original CT library, PriParallel constructs and top-level Parallel constructs create dispatchers. But the lower levels in the hierarchy are again significantly redesigned.

First implementation

Dispatchers are used in the context of their associated PriParallel processes. The PriParallel process will, after it is triggered, ask the associated Dispatcher to activate its subprocesses, or in other words to put them in a ready queue. The Dispatcher is considered to be ready when it contains a Process Thread that can be executed. To ensure that a PriParallel will finish after its subprocesses have finished, it maintains a number of currently active processes (activePTCount). Note that some of the active processes may temporarily be blocked on channels. Every time a PriParallel gets to be executed it will check whether its Dispatcher is ready. If it is ready then it is set to be the current Dispatcher. If it is not ready, the PriParallel is blocked on a semaphore that will be signaled once there are processes ready to run in the Dispatcher. In the code snippet bellow this blocking is hidden in a yield function:

```
void PriParallel::run(void){
   activePTCount = processes->getSize());
   myDispatcher->activateProcesses(this,true);
   while(activePTCount>0){
      Processor::enterAtomic();
      if(myDispatcher->isReady()) {
         Processor::currentDispatcher =
                        myDispatcher;
      }
      else {
          myDispatcher->yield();
      }
      Processor::exitAtomic();
   }
}
```

If a Parallel is a top-level construct and has created a Dispatcher, its run() function behaves the same. Otherwise the same dispatcher will manage the Parallel construct and its subprocesses. In that case, the Parallel construct will only notify the Dispatcher to activate its subprocesses and then it will block until all of its subprocesses finish execution.

In the CT library, an idle task of a nested Dispatcher will give control to the parent dispatcher and only the idle task of the top level process will do idle waiting. Obviously only the idle task of the top-level dispatcher needs to be separate process thread. Idle tasks belonging

to nested dispatchers are not needed; their functionality can be implemented in function calls of passive dispatcher objects.

If not preempted by the release of a higher priority thread, a dispatcher once it takes over, it will give back control to its parent dispatcher only after it does not have process threads to execute. This can lead to starvation of Dispatchers of equal priority.

Changes in class hierarchy

Constructs are essentially processes, which organize execution of other processes. In the CT library, all constructs inherit from the abstract class Process. This class defines only an undefined run() function.

The redesigned CT library recognizes behavior common for all constructs and implements a new parent class (Construct) from which all constructs inherit. Construct class is a kind of Process.

All constructs maintain the list of subprocesses. They also offer a function (exit()) called by subprocesses to notify their parent that they have finished execution.

The Process class is also extended. Now it keeps track of its associated ProcessThread, if any, and desired stack size needed in case this process is executed as a subprocess of a Parallel or a PriParallel construct. This enables a user of the library to specify the stack size based on estimated needs of each process. A Design Space Exploration tool is under development, which should be able to provide estimates of the needed stack sizes for every process.

Centralized Dispatcher

As described in previous subsections, dispatchers are transformed to passive objects, whose sole duty is to maintain ready queues. This form is further transformed to one centralized dispatcher containing a hierarchy of ready queues.

With a centralized dispatcher, there is no more need to switch from one Dispatcher to another. Instead, the ready queue will always get the highest priority thread from the hierarchy of ready queues.

The centralized dispatcher is also more flexible and modular solution. The way the ready queue is organized is localized in one object and not dispersed across the whole library structure. The scheduling mechanism can be changed easily by replacing the internals of the single dispatcher object. Two basic types of ready queues exist: prioritized and FIFO version. However different architectures of ready queue can be built from instances of those two blocks.

Priority comparison in CT library and occam sense

Priorities of two process threads are in the CT library compared by browsing from both of them up through hierarchy of dispatchers until the common parent dispatcher is found. Then, only immediate children of common dispatcher from both ancestor branches are compared. In the following example:

```
PAR P
   PRIPAR P1
      A
      B
      C
   PRIPAR P2
      D
      E
      F
```

In table below, all processes from the example are listed. Associated priorities are represented in binary numbers. The shaded part of the table is scheduled by dispatcher of first PriParallel construct and the rest by the dispatcher of second PriParallel construct. Borders between queues are specified in bold. In this case, one FIFO queue will contain two prioritized queues. Parallel constructs add their process threads in the same FIFO queue where they belong. Priparallel constructs create new prioritized queue for process threads executing its subprocesses.

Priority in Par	Priority in PriPAr	Process
000	000	A
000	001	B
000	010	C
000	000	D
000	001	E
000	010	F

The group of processes A, B and C has strict priority ordering defined. Same goes for a group of processes D, E and F. However since P1 and P2 are considered to have the same priority, their subprocesses are in the occam and the CT library also considered to have the same priorities. For instance, the priority of F is treated as equal when compared to priorities of A, B or C. Due to the usage of PAR there is no strict ordering of priorities in the system.

Components friendly way to perform priority comparison

In the CT library, the released process D will not preempt the running process B, because they are considered to have the same priority.

An alternative way to compare priorities would be to see a priority of some process thread as an array that consist of priorities of all its parent dispatchers and own priority number of process thread from encompassing parent. In this light, considering zero as highest priority process, priorities are arrays of size two: for A and D: {0;0}, for B and E: {0;1}, for C and F{0;2}, Now

priority ordering is fully defined and corresponds better to intuitive expectations.

Of course one can get same priority ordering in the conventional way of comparing priorities by reorganizing the structure of the program:

```
PRIPAR
   PAR
      A
      D
   PAR
      B
      E
   PAR
      C
      F
```

But what if the P1 and P2 are the IP blocks, commercial off the shelf components, that internally distinguish several priority levels- e.g. hard real-time, several soft and not real-time level. Dividing components to pieces in order to have better structure of the program would not be allowed in that case. If P1 and P2 are equally important components it is logical to organize them in a PAR construct. The expected behavior would be that soft or not real-time priority level processes from P2 cannot preempt hard real-time processes of component P2. The priority of (implivet) process thread should in this case include also all the priorities of its parent dispatchers.

The table containing total ready queue of the system should be rearranged according to the changed interpretation of priorities, see the table on the next page.

Priority in Par	Priority in PriPAr	Process
000	000	A
000	000	D
000	001	B
000	001	E
000	010	C
000	010	F

In this case, the hierarchy of ready queues that mimics the hierarchy of PriParallel constructs seems not to be perfect backbone to organize the ready queues. Superior structure of ready queues can be organized as in the following table:

Priority in Par	Priority in PriPAr	Process
000	000	A
000	000	D
000	001	B
000	001	E
000	010	C
000	010	F

In this case, there is a strict hierarchy of prioritized queues. Every prioritized queue maintains one FIFO queue that handles all the Process Threads having the same priority array as it does.

When the hierarchy of ready queues is maintained in the single object (centralized dispatcher) it is very easy to rearrange hierarchy in order to change the way priorities are assigned. Described component-friendly way of priority comparison is implemented. Conditional compiling on few key points allows users to choose one of the two or more implemented scheduling mechanisms in a very simple way.

VI. Conclusions and Future work

Although core parts of the CT library are redesigned, there is still much work to be done. Future work will include redesigning the communication model for external events (handling device drivers and remote communication), adding support for barrier synchronization, and redesigning timer support.

VII. Acknowledgments

The contribution of Gerald Hilderink to this project is substantial. Gerald came up with the idea and implemented a first version of the occam-like kernel library for the real-time control systems application area.

The authors use this opportunity to thank the other present and ex members of our embedded team: P.M. Visser, D. Jovanovic, G. Liet, T. van Engelen and M. Groothuis. All of them had given in several occasions valuable feedback on the subject.

REFERENCES

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

[2] C. A. R. Hoare, *Communicating Sequential Processes*: Prentice Hall, 1985.

[3] G. C. Buttazzo, *Hard real-time computing systems: Predictable Scheduling Algorithms and Applications*. Pisa, Italy: Kluwer Academic Publishers, 2002.

[4] INMOS, *occam 2 Reference Manual*: Prentice Hall, 1988.

[5] R. Meenakshisundaram, "Transputer Home Page" http://www.classiccmp.org/transputer/, 2004.

[6] 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.

[7] Formal Systems, "CSP Tools" http://www.fsel.com, 2004.

[8] J. Kerridge, "Jon Using occam3 to build large parallel systems: Part 1, occam3 features," *Transputer Communications*, vol. 1, pp. 47-63, 1993.

[9] O. J. Fleming, "Parallel Processing in Control - the transputer and other architectures," in *IEE Computing Series*, vol. 38. London: P. Peregrinus, 1988, pp. 244.

[10] G. H. Hilderink, "Communicating Threads home page: www.ce.utwente.nl/JavaPP,", 2002.

[11] P. H. Welch, "The JCSP Homepage" http://www.cs.kent.ac.uk/projects/ofa/jcsp/, 2004.

[12] N. C. C. Brown and P. H. Welch, "An Introduction to the Kent C++CSP Library", In J. F. Broenink and G. H. Hilderink, Eds., *Communicating Process Architectures 2003*. Enschede, Netherlands, 2003.

[13] J. P. E. Sunter, *Allocation, Scheduling and Interfacing in Real-time Parallel Control Systems*, PhD thesis, Faculty of Electrical Engineering, University of Twente, Enschede, Netherlands, 1994.