LUNA: Hard Real-Time, Multi-Threaded, CSP-Capable Execution Framework

M. M. BEZEMER, R. J. W. WILTERDINK and J. F. BROENINK

Control Engineering, Faculty EEMCS, University of Twente, P.O. Box 217 7500 AE Enschede, The Netherlands.

{M.M.Bezemer, J.F.Broenink} @utwente.nl

Abstract. Modern embedded systems have multiple cores available. The CTC++ library is not able to make use of these cores, so a new framework is required to control the robotic setups in our lab. This paper first looks into the available frameworks and compares them to the requirements for controlling the setups. It concludes that none of the available frameworks meet the requirements, so a new framework is developed, called LUNA. The LUNA architecture is component based, resulting in a modular structure. The core components take care of the platform related issues. For each supported platform, these components have a different implementation, effectively providing a platform abstraction layer. High-level components take care of platform-independent tasks, using the core components. Execution engine components implement the algorithms taking care of the execution flow, like a CSP implementation. The paper describes some interesting architectural challenges encountered during the LUNA development and their solutions. It concludes with a comparison between LUNA, C++CSP2 and CTC++. LUNA is shown to be more efficient than CTC++ and C++CSP2 with respect to switching between threads. Also, running a benchmark using CSP constructs, shows that LUNA is more efficient compared to the other two. Furthermore, LUNA is also capable of controlling actual robotic setups with good timing properties.

Keywords. CSP, framework architecture, hard real-time, performance comparison, rendezvous communication, scheduling, threading.

Introduction

Context

Nowadays, many embedded systems have multiple cores at their disposal. In order to be able to run more challenging (control) algorithms, embedded control software should be able to make use of these extra cores. Developing complex concurrent software tends to become tedious and error-prone. CSP [1] can ease such a task. Especially in combination with a graphical modeling tool [2], designing such complex system becomes easier and the tool could help in reusing earlier developed models. CTC++ [3] is a CSP based library, providing a hard real-time execution framework for CSP based applications.

When controlling robotic setups, real-time is an important property. There are two levels of real-time: hard real-time and soft real-time. According to Kopetz [4]: "If a result has utility even after the deadline has passed, the deadline is classified as *soft* (. . .) If a catastrophe could result if a deadline is missed, the deadline is called *hard*".

Figure 1 shows the layered design, used in our Control Engineering group, for embedded software applications connected to actual hardware. Each layer supports a type of real-time,

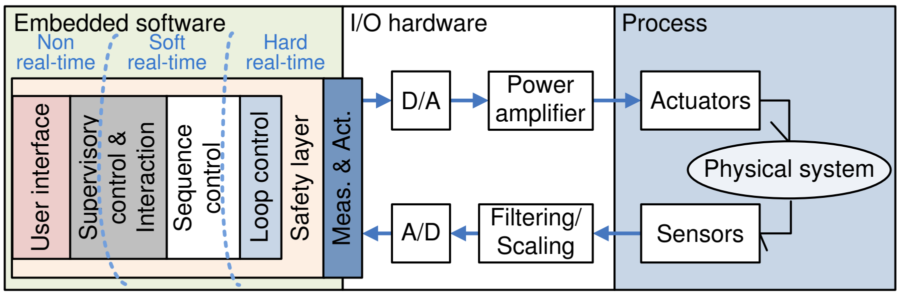

Figure 1. Software architecture for embedded systems [5].

varying from non real-time to hard real-time. The ‘Loop control’ is the part of the application responsible for controlling the physical system and it is realised in a hard real-time layer. The hard real-time layer has strict timing properties, guaranteeing that given deadlines are always met. If this for whatever reason fails, the system is considered unsafe and catastrophic accidents might happen with the physical system or its surroundings due to moving parts. The soft real-time layer tries to meet its deadlines, without giving any hard guarantees. If the design is correct nothing serious should happen in case such a deadline is not met. This layer can be used for those parts of the application which are more complex and require more time to run its tasks, like algorithms which map the environment, plan future tasks of the physical system or communicate with other systems. The non real-time layer does not try to meet any deadlines, but provides means for long running tasks or for an user interface. The left-over resources of the system are used for these tasks, without giving any guarantees of the availability of them.

Robotic and mechatronic setups like the ones in our lab require a hard real-time layer, since it is undesirable for the actual setups to go haywire. The use of Model Driven Development (MDD) tools makes developing for complex setups a less complex and more maintainable task [6]. For the multi-core or multi-CPU embedded platforms, we would like to make use of these extra resources. Unfortunately, the CTC++ library, as it is, is not suitable for these platforms, as it can only use one core or CPU. This paper evaluates possibilities to overcome this problem.

The requirements for a suitable framework that can be used for robotic and mechatronic setups are:

- *Hard real-time*. This incorporates that the resulting application needs to be deterministic, so it is possible to guarantee that deadlines are always met. The framework should provide a layered approach for such hard real-time systems (see Figure 1).
- *Multi-platform*. The setups have different kind of hardware platforms to run on, like PowerPC, ARM or x86 processors. Also different operating systems should be supported by the framework.
- *Thread support*. In order to take advantage of multi-core or multi-CPU capable target systems.
- *Scalability*. All kind of setups should be controlled: From the big robotic humanoids in our lab to small embedded platforms with limited computer resources.
- *CSP execution engine*. Although, it should *not* force the use of CSP constructs when the developer does not want it, as this might result in not using the framework at all.
- *Development time*. The framework should decrease the development time for complex concurrent software.
- *Debugging and tracing*. Provide good debugging and tracing functionality, so developed applications using the framework can be debugged easily and during development unexpected behaviour of the framework can be detected and corrected. Real-time logging functionalities could preserve the debug output for later inspection.

The CTC++ library meets most requirements, however as mentioned before, it does not have thread support for multi-core target systems. It also has a tight integration with the CSP execution engine, so it is not possible to use the library without being forced to use CSP as well. This is an obstacle to use the library from a generic robotics point of view and results in ignoring the CTC++ library altogether, as is experienced in our lab. A future framework should prevent this tight integration. By adding a good MDD tool to the toolchain, the robotic oriented people can gradually get used to CSP.

It might seem logical to perform a major update to CTC++. But unfortunately the architecture and structure of the library became outdated over the years, making it virtually impossible to make such major changes to it. So other solutions need to be found to solve our needs.

Existing Solutions

This section describes other frameworks, which could replace the CTC++ library. For each framework the list with requirements is discussed to get an idea of the usability of the framework.

A good candidate is the C++CSP2 library [7] as it already has a multi-threaded CSP engine available. Unfortunately it is not suitable for hard real-time applications controlling setups. It actively makes use of exceptions to influence the execution flow, which makes a application non deterministic. Exceptions are checked at run-time, by the C++ run-time engine. Because the C++ run-time engine has no notion of custom context switches, exceptions are considered unsafe for usage in hard real-time setups. Also as exceptions cannot be implemented in a deterministic manner, as they might destroy the timing guarantees of the application. Exceptions in normal control flow also do not provide priorities which could be set for processes or groups of processes. This is essential to have hard, soft and non real-time layers in a design in order to meet the scheduled deadlines of control loops. And last, it makes use of features which are not commonly available on embedded systems. On such systems it is common practice to use the microcontroller C library (uClibc) [8], in which only commonly used functionality of the regular C library is included. Most notably, one of the functionalities which is not commonly included in uClibc is *Thread Local Storage*, but is used by C++CSP2.

Since Java is not hard real-time, for example due to the garbage collector, we did not look into the Java based libraries, like JCSP [9]. Although, there is a new Java virtual machine, called JamaicaVM [10], which claims to be hard real-time and supporting multi-core targets. Nonetheless, JCSP was designed without hard real-time constraints in mind and so may not be suitable for hard real-time.

Besides these specific CSP frameworks, there are non-CSP-based frameworks to which a CSP layer might be added. OROCOS [11] and ROS [12] are two of these frameworks and both claim to be real-time. But both will not be able to run hard real-time 1KHz control loops on embedded targets which are low on resources. Their claim about being real-time is probably true when using dedicated hardware for the control loops, which are fed by the framework with ‘setpoints’. Basically, the framework is operating at a soft real-time level, since it does not matter if a setpoint arrives slightly late at the hardware control loop. In our group we like to design the control loops ourselves and are not using such hardware control loop solutions. Furthermore, it is impossible to use formal methods to confirm that a complex application, using one of these frameworks, is deadlock or livelock free, because of the size and complexity of these frameworks [13].

Based on the research performed on these frameworks, we have decided to start over and implement a completely new framework. Available libraries, especially the CTC++ and C++CSP2 libraries, are helpful for certain constructs, ideas and solutions. The new framework can reuse these useful and sophisticated parts, to prevent redundant work and knowledge being thrown away. After implementing the mentioned requirements, it should be able to keep up with our future expansion ideas.

Outline

The next section describes the general idea behind the new framework, threading, the CSP approach, channels and alternative functionality. Section 2 compares the framework with the other related CSP frameworks mentioned earlier, for some timing tests and when actually controlling real setups. In the next section, the conclusions about the new framework are presented. And the last section discusses future work and possibilities.

1. LUNA Architecture

The new framework is called LUNA, which stands for ‘LUNA is a Universal Networking Architecture’. A (new) graphical design and code generation tool, like gCSP [14], is also planned, tailored to be compatible with the LUNA. This MDD tool will be called Twente Embedded Real-time Robotic Application (TERRA). It is going to take care of model optimisations and by result generating more efficient code, in order to reduce the complexity and needs of optimisations in LUNA itself.

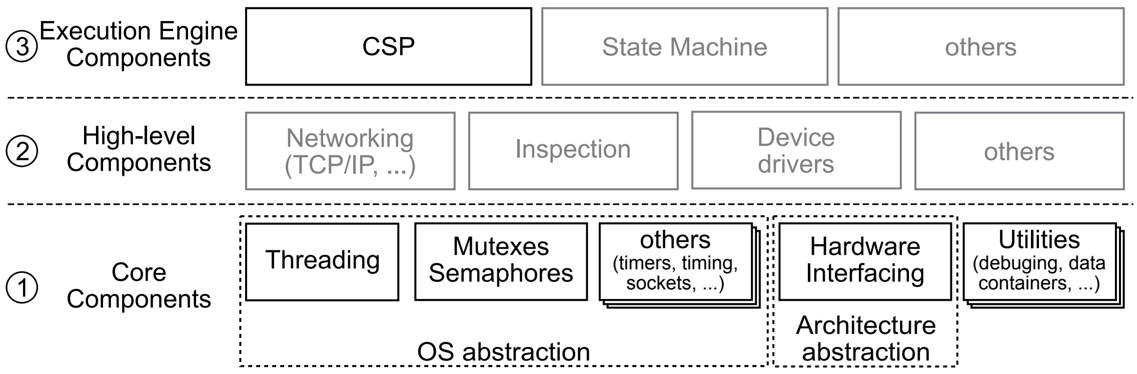

Figure 2. Overview of the LUNA architecture.

LUNA is a component based framework that supports multiple target platforms, currently planned are QNX, RTAI and Xenomai. To make development more straightforward, Linux and Windows will also be supported as additional platforms. Figure 2 shows the overview of the LUNA components and the levels they are on. The gray components are not implemented yet, but are planned for future releases.

The *Core Components* (1) level contains basic components, mostly consisting of platform supporting components, providing a generic interface for the platform specific features. OS abstraction components are available to support the target operating system (OS), like threading, mutexes, timers and timing. The architecture abstraction components provide support for features specific to an architecture (or hardware platform), like the support for (digital) input and output (I/O) possibilities. Other components can make use of these core components to make use of platform specific features without knowledge of the actual chosen platform. Another group of core components are the utility components, implementing features like debugging, generic interfaces and data containers.

The next level contains the *High-level Components* (2). These are platform independent by implementing functionality using the core components. An example is the Networking component, providing networking functionality and protocols. This typically uses a socket component as platform-dependent glue and build (high-level) protocols upon these sockets.

The *Execution Engine Components* (3) implement (complex) execution engines, which are used to determine the flow of the application. For example a CSP component provides constructs to have a CSP-based execution flow. The CSP component typically uses the core components for threading, mutexes and so on and it uses high-level components like networking to implement networked rendezvous channels.

Components can be enabled or disabled in the framework depending on the type of application one would like to develop, so unused features can be turned off in order to save resources. Since building LUNA is complex due to the component based approach and the variety of supported platforms, a dedicated build system is provided. It is heavily based on the OpenWrt buildroot [15,16].

The initially supported platform is QNX [17], which is a real-time micro-kernel OS. QNX natively supports hard real-time and rendezvous communication. This seemed ideal to start with, relieving the development load for an initial version of LUNA. As QNX is POSIX compliant, a QNX implementation of LUNA would result in supporting other POSIX compliant operating systems as well. Or, at least it would support parts of the OS which are compatible, as not many operating systems are fully POSIX compliant.

1.1. Threading Implementation

LUNA supports OS threads (also called kernel threads) and User threads to be able to make optimal use of multi-core environments. OS threads are resource-heavy, but are able to run on different cores and User threads are light on resources, but must run in a OS thread and are thus running on the same core as the OS thread. A big advantage of using OS threads is the preemptive capabilities of these threads: Their execution can be forcefully paused anywhere during its execution, for example due to a higher priority thread becoming ready. User threads can only be paused at specified moments, if such a moment is not reached, for example due to complex algorithm calculations, other User threads on the same OS thread will not get activated. Combining resource-heavy OS threads and non preemptive capable User threads results in a hybrid solution. This allows for constructing groups of threads which can be preempted but are not too resource-heavy.

As the term already implies, the OS threads are provided and maintained by the OS. For example, the QNX implementation uses the POSIX thread implementation provided by QNX and for Windows LUNA would use the Windows Threads. Therefore, the behaviour of an OS thread might not be the exactly the same for each platform.

The User threads are implemented and managed by LUNA, using the same principles as [7,18], except the LUNA User threads are not run-time portable to other OS threads. There is no need for it and this will break hard real-time constraints.

Figure 3 shows the LUNA threading architecture. Two of the components levels of Figure 2 are visible, showing the separation of the threading implementation and the CSP implementation.

UThreadContainer (UTC) and OSThread are two of the available thread types, both implementing the IThread interface. This IThread interface requires a Runnable, which acts as a container to hold the actual code which will be executed on the thread. The CSP functionality, described in more detail in the next section, makes use of the Runnable to provide the code for the actual CSP implementation.

To make the earlier mentioned hybrid solution work, each OS thread needs its own scheduler to schedule the User threads. This scheduling mechanism is divided into two objects:

1. the UTC which handles the actual context switching in order to activate or stop a User thread.

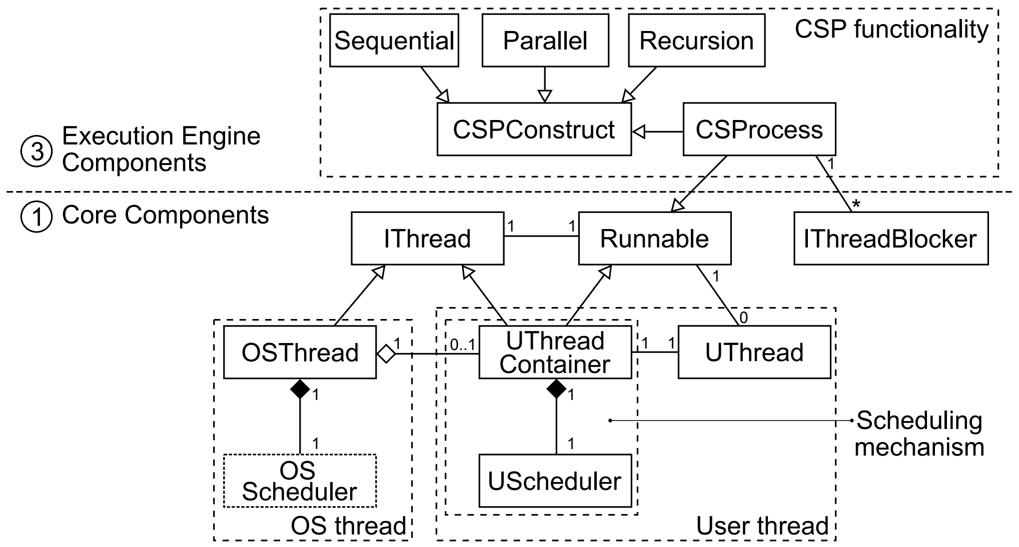

Figure 3. UML diagram of threads and their related parts.

2. the UScheduler which contains the ready and blocked queue and decides which User thread is the next to become active.

The UTC also contains a list with UThreads, which are the objects containing the 'context' of a User thread: the stack, its size and other related information. Besides this context relation data, it also contains a relation with the Runnable which should be executed on the User thread.

For the CSP functionality a 'separation of concerns' approach is taken for the CSP processes and the threads they run on. The CSP processes are indifferent whether the underlying thread is an OS thread or a User thread, which is a major advantage when running on multi-core targets. This approach can be taken a step further in a distributed CSP environment where processes are activated on different nodes. This will also facilitate deployment, seen from a supervisory control node. Due to this separation, it is also possible to easily implement other execution models.

The figure shows that the Sequential, Parallel and Recursion processes are not inheriting from CSProcess but from CSPConstruct. The CSPConstruct interface defines the activate, done and exit functions and CSProcess defines the actual run functionality and context blocking mechanisms. Letting the processes inherit from CSPConstruct is an optimisation: This way they do not require context-switches because their functionality is placed in the activate and done functions, which is executed in the context of its parent respectively child threads. The Alternative implementation still is a CSProcess, because it might need to wait on one of its guards to become ready and therefore needs the context blocking functionality of the CSProcess.

The UTC implements the Runnable interface so that it can be executed on an OS thread. When the UTC threading mechanism starts, it switches to the first User thread as a kickstart for the whole process. When the User thread is finished, yields or is explicitly blocked, the UTC code switches to the next User thread which is ready for execution. Due to this architectural decision, the scheduling mechanism is not running on a separate thread, but makes use of the original thread, in between the execution of two User threads.

During tests, the number of threads was increased to 10,000 without any problems. All threads got created initially and they performed their task: increase a number and print it. After executing its task, each thread was properly shutdown.

1.2. LUNA CSP

Since LUNA is component based, it is possible to add another layer on top of the threading support. Such a layer is the support layer for a CSP-based execution engine. It is completely separated from the threading model, so it will run on any platform that has threading support within LUNA.

Each CSP process is mapped on a thread. Because of the separation of CSP and the threading model, the CSP processes are indifferent whether the underlying thread is an OS thread or a User thread, which is a major advantage when running on multi-core targets. This will also facilitate code generation, since code generation needs to be able to decide how to map the CSP processes on the available cores in an efficient way without being limited by thread types.

Figure 4 shows the execution flow of three CSProcess components, being part of this greater application:

```
P = Q || R || S
Q = T; U
```

Process **P** is a parallel process and has some child processes, of which process **Q** is one. Process **Q** is a sequential process and also has some child processes. Process **T** is one of these child processes and it does not have any child processes of its own.

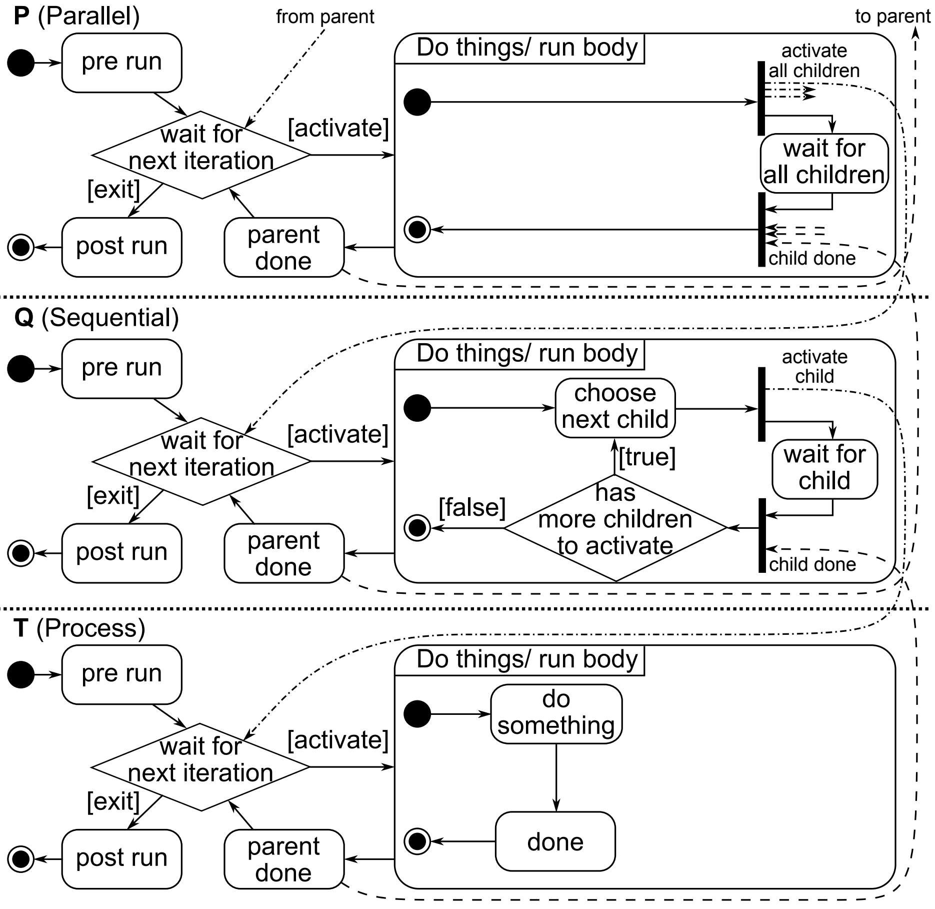

Figure 4. Flow diagram showing the conceptual execution flow of a CSProcess.

First, the *pre run* of all processes is executed, this can be used to initialise the process just before running the actual semantics of the CSProcess. Next the processes are waiting in *wait for next iteration* until they are allowed to start their *run body*. After all processes have executed their *pre run* the application itself is really started, so the *pre run* does not have to be deterministic yet. The *post run* of each process is executed, when the process is shutdown, normally when the application itself is shutdown. It gives the processes a chance to clean up the things they initialised in their *pre run*.

In this example, **P** will start when it is activated by its parent. Due to the parallel nature of the process, all children are activated at once and next the process will wait until all children are done before signalling the parent that the process is finished. Process **Q** is only one of the processes that is activated by **P**. **Q** will activate only its first child process and waits for it until it is finished, because **Q** is a sequential process. If there are more children available, the next one is activated and so on. **T** is just a simple code blob which needs to be executed. So at some point it is activated by **Q**, it executes its code and sends signal back to **Q** that it is finished. Same goes for **Q**, when all its child processes are finished, it sends back a signal to **P**, telling it is finished.

Due to this behaviour, the CSP constructs are implemented decentralised by the CSProcesses, instead of implemented by a central scheduler. This results in a simple generic scheduling mechanism, without any knowledge of the CSP constructs. Unlike CTC++, which has a scheduler implemented that has knowledge of all CSP constructs in order to implement them and run the processes in the correct order.

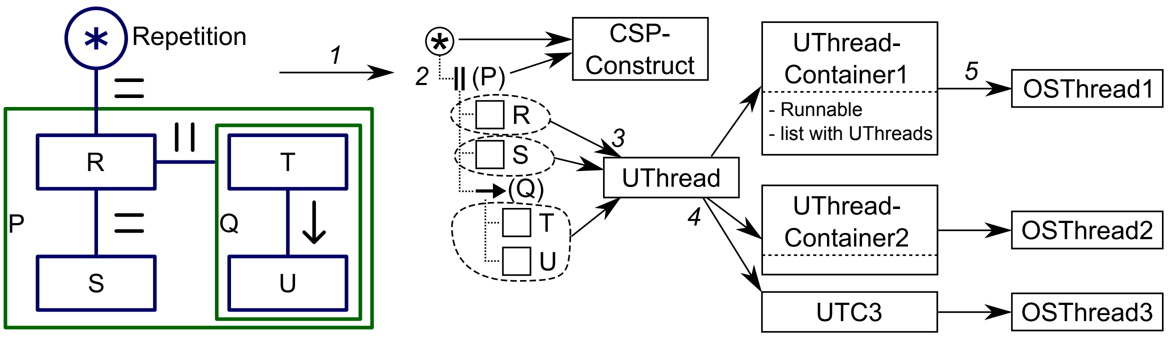

Figure 5. Steps from a model to a LUNA based mapping on OS threads.

Since the CSP processes are indifferent to the type of thread they run on and how they are grouped on OS threads, LUNA needs to provide a mechanism to actually attach these processes to threads. When looking at a gCSP model (left-most part of the figure), a compositional hierarchy can be identified in the form of a tree (middle part of the figure). The MDD tool has to map the processes onto a mix of OS and User threads using the compositional information and generate code. Because of the 'separation of concerns' code generation is straightforward as the interoperation of OS and User threads is handled by LUNA. Figure 5 shows the required steps to map the model to OS threads.

First, the model needs to be converted to a model tree (number *1* in the figure). This model-tree contains the compositional relations between all processes. Second, the user (or the modeling tool) needs to group processes (*2*) which are put on the same OS thread, for example criteria for grouping could be processes which heavily rely on communication or try to balance the execution load. Each process is mapped to a UThread object (*3*). Except for the compositional processes mentioned in the previous section, they are mapped onto CSPConstructs. Next, each group of of UThreads is put in an UThreadContainer (UTC) (*4*).

Finally, each UTC is mapped to an OS thread (*5*), so the groups of processes can actually run in parallel and have preemption capabilities. It is clear that making good groups of processes will influence the efficiency of the application, so using an automated tool is recommended [19].

1.3. Channels

One of the initial reasons for supporting QNX was the availability of native rendezvous communication support between QNX threads. This indeed made it easy to implement channels for the OS threads, but unfortunately it was *not* for the User threads. Main problem is that two User threads which want to communicate may be placed on the same OS thread. If one User thread wants to communicate over a rendezvous channel and the other side is not ready, the QNX channel blocks the thread. But QNX does not know about the LUNA implemented scheduler and its User threads, so it blocks the OS thread instead. The other User thread which is required for the communication now never becomes ready and a deadlock occurs. So unfortunately, for communication between User threads on the *same* OS thread the QNX rendezvous channels are not usable and the choice to initially support QNX became less strong.

Figure 6 shows the 2 possible channel types. Channel 1 is a channel between two OS threads. The QNX rendezvous mechanism can be used for this channel. Channel 2a and 2b are communication channels between two User threads; it does not matter whether the User threads are on the same OS thread or not. For this type of channel the QNX rendezvous mechanisms cannot be used as explained earlier, as it could block the OS thread and therefore prevent execution of other User threads on that OS thread. An exception could be made for

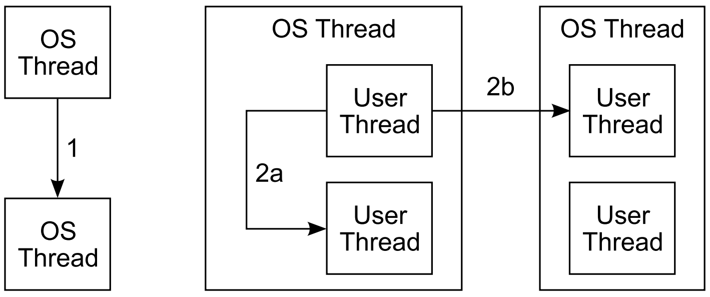

Figure 6. Overview of the different channel situations.

OS threads with one User thread, but such situations are undesired since it is more efficient to directly run code on the OS thread without the User thread in between. Guarded channels are also not supported by QNX, so for this type of channels a custom implementation is also required.

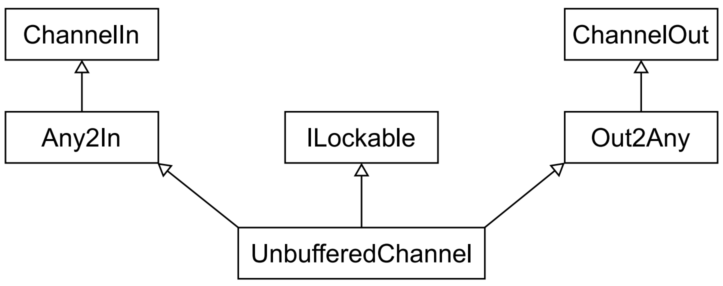

Figure 7. Diagram showing the channel architecture.

Figure 7 shows the architecture of the channel implementation. A channel is constructed modularly: The buffer, Any2In and Out2Any types can be exchanged with other compatible types. The figure shows an unbuffered any-to-any channel, but a buffered any-to-one is also possible, along with all kinds of other combinations.

```
write() {
  ILockable.lock()
  if (isReaderReady()) {
    IReader reader = findReadyReaderOrBuffer()
    transfer(writer, reader)
    reader.unblockContext()
    ILockable.unlock()
  } else {
    setWriterReady(writer)
    ready_list.add(writer)
    writer.blockContext(ILockable)
  }
}
```

Listing 1. Pseudocode showing the channel behaviour for a write action.

Listing 1 shows the pseudocode for writing on a channel. The ILockable interface is used to gain exclusive access to the channel, in order to make it 'thread safe'. Basically, there are two options: Either there is a reader (or buffer) ready to communicate or not. If the reader is already waiting, the data transfer is performed and the reader is unblocked so it can be scheduled again by its scheduler when possible. In the situation that the reader is not available, the writer needs to be added to the *ready_list* of the channel, so the channel knows about the writers which are ready for communication. This list is ordered on process priority. And, the writer needs to be blocked until a reader is present. The same goes for reading a channel, but exactly the other way around.

The *findReadyReaderOrBuffer()* method checks if there is buffered data available, otherwise it calls a *findReadyReader()* method to search for a reader which is ready. The *isReaderReady()* and *findReadyReader()* methods are implemented by the Out2Any block or by a similar block that is used. So depending on the input type of the channel, the implementation is quite simple when there is only one reader allowed on the channel or more complex when multiple readers are allowed. The *transfer()* method is implemented by the (Un)bufferedChannel and therefore is able to read from a buffer or from an actual reader depending on the channel type.

LUNA supports communication between two User threads on the same OS thread by a custom developed rendezvous mechanism. When a thread tries to communicate over a channel and the other side is not ready, it gets blocked using the IThreadBlocker (see Figure 3). By using the IThreadBlocker interface, the thread type does not matter since the implementation of this interface is dependent on the thread type. For User threads, the scheduler puts the current thread on the blocked queue and activates a context-switch to another User thread which is ready. This way the OS thread is still running and the User thread is blocked till the channel becomes ready and the scheduler activates it. And for OS threads, it uses a semaphore to completely block the OS thread until the channel is ready.

As mentioned in the start of this section, there are different implementations of channels: the QNX implementation used for communication between OS threads and the LUNA implementation for communication between User threads and/or OS threads. It would be cumbersome for a developer to have to remember to choose between these types, especially when the User threads are not yet mapped to their final OS threads. So a channel factory is implemented in LUNA. When all CSP processes are mapped on their threads, this factory can be used to determine what types of channels are required. Having the information of the type of threads to map the CSP processes on is sufficient to determine the required channel implementation. At run-time, before the threads are activated, the factory needs be invoked to select a correct implementation for each channel. If a developer (or code generation) moves a CSP process to another OS thread, the factory will adapt accordingly, using the correct channel implementation for the new situation.

1.4. Alternative

The Alternative architecture is shown in Figure 8. It is a CSProcess itself, but it also has a list of other CSProcesses which implement the IGuard interface. Alternative uses the list when it is activated and will try to find a process which meets its IGuard conditions. Currently, the only guarded processes that are available are the GuardedWriter and GuardedReader processes. But others might be added, as long as they implement the IGuard interface.

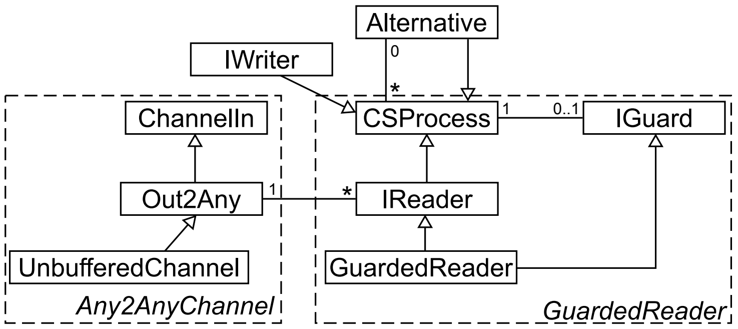

Figure 8. Diagram showing the relations for the Alternative architecture.

In the case of channel communication, it first checks if a reader or writer is guaranteed to perform channel communication without blocking and makes sure this guarantee stays intact. Next, it performs the communication itself. The Alternative implements a sophisticated protocol in order to make sure the communication is guaranteed, even though different threads are part of the communication or some of the processes on the channel might be not guarded.

First in Figure 9 a situation is shown, where a guarded reader gains access on a channel, but blocks when it should actually read the contents, as another reader came in-between. Some of the objects in Figure 8 are grouped by the dashed boxes, they are shown in Figure 9 as a single object to keep things simple.

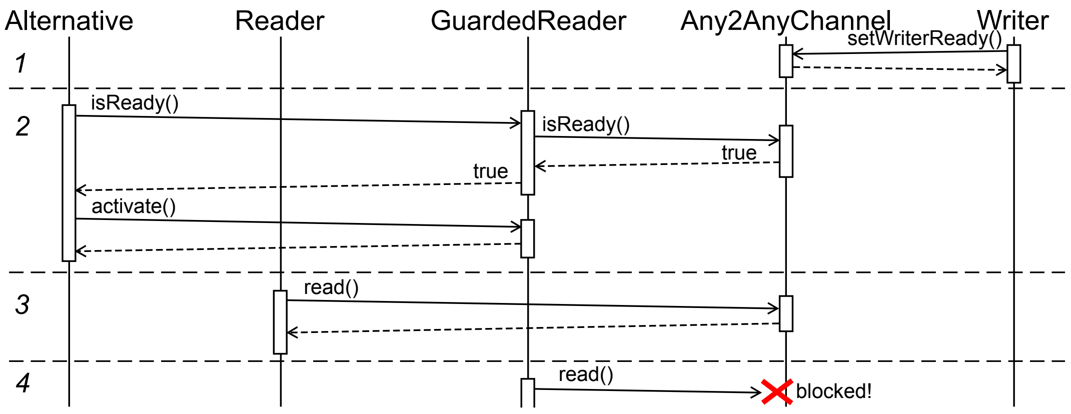

Figure 9. Sequence diagram showing a situation were a guarded reader blocks.

Assume we have an any-to-any channel, which has a writer waiting to communicate (*1* in the figure). The Alternative is activated (*2*) and checks if the GuardedReader is ready. The GuardedReader is only ready if there is a writer or buffer waiting to communicate, so it checks with the channel. When the GuardedReader indeed is ready, it gets activated so it can be scheduled by the scheduler to actually perform the communication.

Unfortunately before the communication takes place, another Reader is activated and wants to communicate on the channel as well (*3*), since there is a writer present the communication takes place. Later, the GuardedReader is activated (*4*), but the writer is not available anymore and the GuardedReader is blocked, even though it gained access to the channel through the Alternative.

To prevent such behaviour a more sophisticated method is used, shown in Figure 10. This example describes a situation where both channel ends are guarded to be able to describe the protocol completely. Whether this situation is used in real applications or not is out of scope.

Again the writer registers at the channel, telling that it is ready to write data (*1*). There is no reader available yet, so the write is put in the *ready_list* and gets a *false* as result. Next, Alternate2 continues to look for a process which can be activated, but this is not interesting for the current situation.

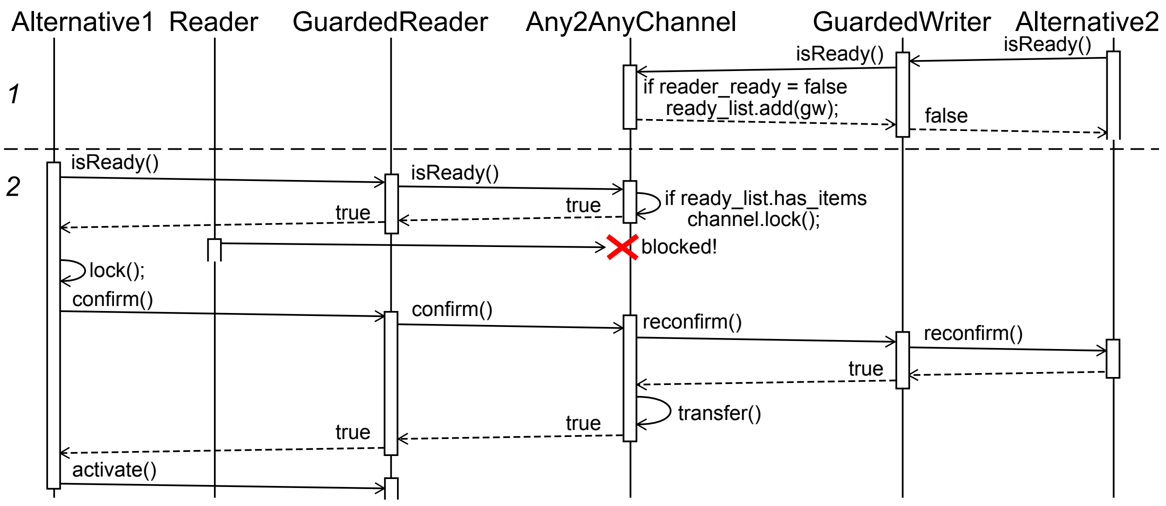

Figure 10. Sequence diagram showing the correct situation.

Alternate1 checks whether the GuardedReader is ready or not, when it becomes active (*2*). Since the *ready_list* has items on it, the channel is ready for communication. To prevent that other readers are interfering with our protocol, the channel gets locked. If Reader wants to read from the channel it gets blocked due to the lock. This is in contrast with the previous example, where the GuardedReader got wrongly blocked.

When the *isReady()* request returns positive, the Alternative1 checks whether another, previously *isReady()* requested, guard has not been reconfirmed. If this is not the case, it will *lock()* the Alternative for exclusive reconfirm request, preventing other guards taking over the current communication.

Before the actual transfer, Alternative1 needs to check whether the GuardedWriter is still ready to write. It might be possible that Alternate2 found another process to activate and the GuardedWriter is not ready anymore. Using the *confirm()* method, Alternative1 asks the channel for this and the channel forwards the question at Alternate2 via GuardedWriter with the *reconfirm()* method. Assuming that the GuardedWriter is still ready, the channel directly performs the *transfer* of data. This is not necessary, but is more efficient as the channel becomes available for other communications earlier.

In the end, Alternative1 revokes the *isReady()* requests of its other guarded processes, since a process was chosen, and it activates GuardedReader. For this example situation it is unnecessary, since the transfer is completed already, but for other (non reader/writer) processes it is required to run the guarded process code. Also, the GuardedReader might be used to activate a chain of other processes.

The described alternative sequence of Figure 10 has been tested for some basic use cases. Although it is not formally proven, it is believed that this implementation will satisfy the CSP requirements of the alternative construction.

2. Results

This section shows some of the results of the tests performed on/with the LUNA framework. The tests compare LUNA with other CSP frameworks, to see how the LUNA implementation performs.

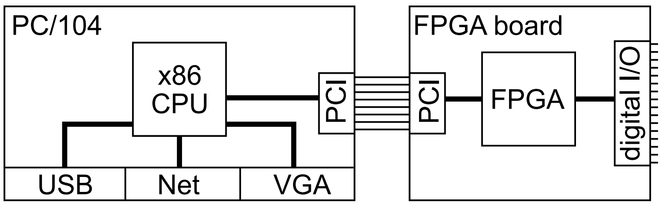

Figure 11. Overview of the used test setup.

All tests in this section are performed on a embedded PC/104 platform with 600 MHz x86 CPU as shown in Figure 11. It is equipped with an FPGA based digital I/O board to connect it with actual hardware when required for the test. While implementing and testing LUNA, QNX seemed to be slower than Linux. To keep the test results comparable, all presented tests are executed under QNX (version 6.4.1) and compiled with with the corresponding qcc (version 4.3.3) with the same flags (optimisation flag: -O2) enabled.

2.1. Context-switch Speed

After the threading model was implemented, a context-switch speed test was performed to get an idea of the efficiency of the LUNA architecture and implementation. To measure this speed, an application was developed consisting of two threads switching 10,000 times. The execution times were measured and the average switching time was calculated to get a more precise context-switching time. Table 1 shows these times.

Table 1. Context-switch speeds for different platforms.

Platform	OS thread (μs)	User thread(μs)
CTC++ ‘original’	-	4.275
C++CSP2	3.224	3.960
CTC++ QNX	3.213	-
LUNA QNX	3.226	1.569

The *CTC++ ‘original’* row shows the test results of the original CTC++ library compiled for QNX. It is not a complete QNX implementation, but only the required parts for the test are made available. In order to be able to compile the CTC++ library for QNX, some things needed to change:

- The `_setjmp/_longjmp` implementation used when switching to another User thread. The Stack Pointer (SP) was changed to use the correct field for QNX.
- Linux does not save the signal mask by default when executing `_setjmp` and `_longjmp`. QNX does, which slows down the context switches considerable. Therefore, the ‘_’ versions of `setjmp` and `longjmp` are used for the QNX conversion.
- The compiler and its flags in order to use the QNX variants.
- The inclusions of the default Linux headers are replaced with their QNX counterparts.

- Some platform-dependent code did not compile and is not required to be able to run the tests, so it was removed.

To use the C++CSP2 library with QNX, the same changes were made as for CTC++ library except the SP modification, as it was not required for C++CSP2. As mentioned the `_setjmp`/`_longjmp` are used for the quick conversion to QNX, although the library already used `_longjmp`, but not `_setjmp`. This might indicate that the author knew of this difference and intended different behaviour. The QNX implementation for the C++CSP2 library is also not complete, only the required parts are tested, all other parts are not tested for compatibility. For the test a custom application was created as the provided C++CSP2 test suite did not contain a pure context switching test.

CTC++ QNX [20] is an initial attempt to recreate the CTC++ library for QNX. It was not completely finished, but all parts needed for the commstime benchmark are available.

LUNA QNX is the new LUNA framework compiled with the QNX platform support enabled. For other platforms the results will be different, but the same goes for the other libraries.

The *OS thread* column shows the time it takes to switch between two OS threads. The *User thread* column shows the time it takes to switch between two User threads placed on the same OS thread.

For LUNA it is clear that the OS thread context-switches are slower than the User thread switches, which is expected and the reason for the availability of User threads. All 3 OS thread implementations almost directly invoke the OS scheduler and therefore have roughly the same context-switch times.

A surprising result is found for C++CSP2: The OS thread context-switch time is similar with the User thread time. The User threads are switched by the custom scheduler, which seems to contain a lot overhead, probably for the CSP implementation. Expected behaviour is found in the next test, when CSP constructs are executed. In this test the custom scheduler gets invoked for the OS threads as well, resulting in an increase of OS context switch time. In this situation the User threads become much faster than the OS threads.

The context-switch time for the LUNA User threads is much lower compared to the others. The LUNA scheduler has a simple design and implementation, as the actual CSP constructs are in the CSProcess objects themselves. This approach pays off when purely looking at context-switch speeds. The next section performs a test that actually runs CSP constructs, showing whether it also pays off for such a situation.

2.2. *Commstime Benchmark*

To get an better idea of the scheduling overhead, the commstime benchmark [21] is implemented, as shown in Figure 12. This test passes a token along a circular chain of processes. The Prefix process starts the sequence by passing the token to Delta, which again passes it on to the Prefix via the Successor process. The Delta process also signals the TimeAnalysis process, so it is able to measure the time it took to pass the token around. The difference between this benchmark and the context-switch speed test, is that in this situation a scheduler is required to activate the correct CSP process depending on the position of the token.

Table 2 shows the cycle times for each library for the commstime benchmark. The commstime tests are taken from the respective examples and assumed to be optimal for their CSP implementation. LUNA QNX has two values: the first is for the LUNA channel implementation and the second value for the QNX channel implementation. It is remarkable that the QNX channels are slower than the LUNA channels. This is probably due to the fact the QNX channels are always any-to-any and the used LUNA channels one-to-one. The amount of context-switches of OS threads is unknown, since the actual thread switching is handled

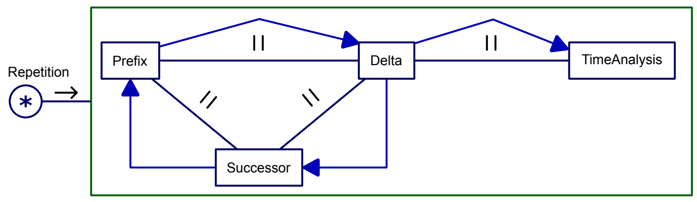

Figure 12. Model of the commstime benchmark.

by the OS scheduler having preemption capabilities and there is no interface to retrieve this data.

Table 2. Overhead of the schedulers implemented by the libraries for their supported thread types.

Platform	Thread type	Cycle time (μs)	# Context-switches	# Threads
CTC++ 'original'	User	40.76	5	4
C++CSP2	OS	44.59	-	4
	User	18.60	4	4
CTC++ QNX	OS	57.06	-	4
LUNA QNX	OS	28.02 / 34.03	-	4
	User	9.34	4	4

Normally, the library is used with modeling tools in combination with code generation. This would result in a different implementation of the commstime benchmark. In general, the readers and the writers become separate processes, instead of integrated within the Prefix, Delta, Successor and TimeAnalysis processes. For example, in this situation the Successor is implemented using a sequential process containing a reader, an increment and a writer process.

Table 3 shows the results when gCSP in combination with code generation is used to design the commstime benchmark application. gCSP code generation is only available for CTC++, so for LUNA the CTC++ code is rewritten manually as if it would have been generated.

Table 3. Commstime results when using MDD tools to create the test.

Platform	Thread type	Cycle time (μs)	# Context-switches	# Threads
CTC++ 'original'	User	88.89	10	6
C++CSP2	OS	12554.95	-	+15
	User	12896.22	19	+15
CTC++ QNX	OS	219.71	-	6
LUNA QNX	OS	93.23 / 99.62	-	10
	User	29.87	14	10

The implementation of the C++CSP2 test was somewhat different compared to the other implementations. Since C++CSP2 threads are destroyed when one cycle is done, they need to be recreated for each cycle. The processes added to sequential process, for example in the Successor, cannot contain a loop, since this would prevent the execution of the second and the third process because those processes need to wait on preceding processes. Due to this limitation, the C+CSP2 implementation needs to recreate 15 threads each cycle, hence the +15 in the table. The construction and destruction of these threads generates a lot of overhead, resulting in the high cycle times around $12.5ms$. It was not possible to prevent this behaviour when using the 'code generated' code, due to differences in the design ideas behind the libraries.

The table also shows that the close result of the CTC++ 'original' and the CTC++ QNX libraries were accidental. Now the difference is bigger, which is expected since the CTC++ QNX library uses OS threads which have much more overhead compared to the User threads. For the first results, the optimised channels of the QNX variant probably resulted in the small difference between the two.

The benchmark results of LUNA are much better compared to the CTC++ library. Furthermore the LUNA results are better than the C++CSP2 results when looking at Table2. This is due to the efficient context-switches, as described in the previous section. When compensating for the required context-switch times, the results for C++CSP2 and LUNA are similar.

When comparing both tables, it is clear that using MDD tools with code generation results in slower code. For simple applications it is advisable to manually create the code, especially for low-resource embedded systems. When creating a complex application to control a large setup, like a humanoid robot, it saves a lot of development time to make use of the MMD tools. For this 'code generated' results, the LUNA framework has good cycle times, which is encouraging since the planning of TERRA, the new MDD tool, which will feature code generation for LUNA. It is advisable for such an MDD tool to invest effort into optimising code generation to get good performance on the target system.

2.3. Real Robotic Setup

Next, an implementation for a real robotic setup was developed with LUNA, to see whether it is usable in a practical way. To keep things easy for a first experiment, a simple pan-tilt setup is used, with 2 motors and 2 encoders. These 2 degrees of freedom can be controlled using a joystick. The control algorithm of this setup requires about 50 context switches to completely run one cycle.

The CTC++ library already has an implementation for this setup available and a similar implementation was made for LUNA to keep the comparison fair. Real-time logging functionality was added in order to be able to measure timing information and to compare LUNA with the CTC++ library.

Table 4 shows the timing results of LUNA and the CTC++ implementation. The experiments have been performed with $100Hz$ and $1kHz$ sample frequencies, so each control loop cycle should be respectively $10ms$ and $1ms$ long. As the measurements were performed for about 60 seconds, the $100Hz$ measurements resulted in about 6,000 samples and the 1 kHz resulted in about 60,000 samples. The processing time is found by subtracting the idle time from the cycle time. The idle time is calculated by measuring the time between the point where the control code is finished and the point where the timer fires an event for the next cycle.

Table 4. Timing results of the robotic implementation.

	Frequency	Cycle time (ms)			Standard	Processing
Platform	(Hz)	Mean	Min	Max	deviation (μs)	time (μs)
CTC++ 'original'	100	11.00	10.90	11.11	14.8	199.0
	1000	1.18	0.91	2.10	386.5	174.5
	1000.15	1.00	0.91	1.10	20.7	172.5
LUNA QNX User threads	100	10.00	9.93	11.00	39.6	111.6
	1000	1.00	0.80	2.01	35.8	89.3
	1000.15	1.00	0.79	1.21	33.2	87.3
LUNA QNX OS threads	100	10.00	9.97	11.00	39.1	214.3
	1000	1.00	0.96	2.00	14.4	185.6
	1000.15	1.00	0.95	1.05	8.3	190.8

The results show that LUNA performs well within hard real-time boundaries. The mean values are a good match compared to the used frequencies and a low standard deviation value shows that the amount of missed deadlines is negligible.

Due to periodically missed clock ticks, the maximum cycle time of the $1kHz$ measurements is twice the sample time. This phenomenon can be explained by the mismatch between the requested timer interval and the PC/104's hardware timer [22]. The timer can not fire exactly every $1ms$, but instead it fires every $0.999847ms$ and for every 6535 instances the timer will not fire. In the $100Hz$ case this will not be noticed, because the cycle time is large enough and these kind of errors are relatively small.

When looking at the CTC++ 'original' implementation, it is seen that the $100Hz$ results are good as well, although the mean cycle time, shows that the obtained frequency is $90.9Hz$ instead of $100Hz$. Same goes for the $1kHz$ measurement where a $847.5Hz$ frequency was obtained instead. From this it can be concluded that CTC++ has problems to closely provide the requested frequencies. For a frequency of $1kHz$, the standard deviation becomes very large.

A third frequency was also measured, $1000.15Hz$, which is an exact match with the available frequency of the setup. This solves the very large standard deviation and the incorrect mean cycle times for the CTC++ library. It should be noted that this frequency is setup dependent and therefore needs to be measured for each setup separately, in order to gain these good results.

The frequency of $1000.15Hz$ indeed solves the maximum cycle times of LUNA being two periods long. For setups which needs to be extremely accurate this is important, as it can make the difference between an industrial robot moving smoothly or scratching your car. The other values are not much different, showing that LUNA is more robust for all frequencies than the CTC++ library and frequency tuning is not required to get reasonable hard real-time properties.

It is also noticeable that the processing times for the LUNA User threads are lower compared to the CTC++ processing times. Suggesting that the overhead is much lower and that more resources are available for the controlling code. Even the LUNA OS threads processing times are comparable with the CTC++ User thread processing times.

3. Conclusions

Good results are obtained using LUNA, it has fast context-switches and the commstime benchmark is faster than the C++CSP2 and CTC++ implementations.

These benchmark results are good but the main requirement, the real-time behaviour of the library, is much more important when controlling robotic setups. The simple robotic setup indeed performed as expected; it reacts smoothly on the joystick commands. The maximum and minimum cycle time values are close to the (requested) mean cycle time and the standard deviation values are low, showing that the hard real-time properties of LUNA are good.

The choice for QNX is not that obvious anymore when the provided rendezvous channels are only usable between OS threads. Nonetheless, QNX provides a good platform to build a real-time framework, there is enough support from the OS to keep implementation tasks maintainable.

All requirements mentioned in the introduction are met. The first three of them are obvious: LUNA is a hard real-time, multi-platform, multi-threaded framework.

Scalability is also met, even though LUNA was not yet tested with a big (robotic) setup, early scalability tests showed that having 10,000 processes poses no problem.

The CSP execution engine is the only implemented execution engine at the moment. But the requirement to not be dependent on it is met, as it is possible to turn it off and use the User and OS threads in a non CSP related way. Using the provided interface it is also possible to add other execution engines like a state machine execution engine.

Developing applications using LUNA is straightforward, for example one does not need to keep the type of threads and channels in mind while designing the control application. It is possible to just create the CSP processes, connect them with channels and let the LUNA factories decide on the actual implementation types.

Finally, Debugging and Tracing is also not a problem – it is possible to enable the debugging component if required. This component contains means for debugging and tracing the other components as well as the application is being developed. It is also possible to send the debug and trace information over a (local) network to a development PC, in order to have run-time analysis or to store it for off-line analysis.

The required logger does not influence the executing application noticeable as it is a real-time logger. It has predefined buffers to store the debug information and only when there is idle CPU time available, it sends the buffered content over the network freeing up the buffer for new data.

Especially logging the activation of processes is interesting, as this could provide valuable timing information, like the cycle time of a control loop or the jitter during execution. So it is possible to influence the application with external events and directly see the results of such actions. It is also possible to following the execution of the application by monitoring the states (running, ready, blocked, finished) of the processes. This information could also be fed back to the MDD tool, in order to show these states in the designed model of the application.

For *future work* an implementation for Linux (and Windows) would be convenient. It is much faster to try out new implementation on the development PC than on a target. Of course this requires more work, hence the choice to support QNX first, but it certainly pays off by reducing development time. The flexibility to easily move processes between the groups of OS and User threads reduces development time even more, as the developer does not required to change his code when moving processes.

Building the simple robotic setup took some time. There are only about 51 processes to control this setup. Of course this could be less, but it takes too much time to develop controller applications by hand, so code generation for LUNA is required. In order to attract users to start using LUNA – and also for educational purposes – code generation is required. So, soon after LUNA evolves into an initial and stable version, TERRA needs to be built as well to gain these advantages and properly use LUNA.

When TERRA and code generation are available, algorithms to optimise the model for a specified target with known resources can be implemented. Before code generation, these algorithms [19] can schedule the processes automatically in an optimal manner for the available resources. These scheduling algorithms are also interesting for performing timing analysis of the model, in order to estimate whether the model will be able to run real-time with the available resources.

To see whether LUNA is capable of controlling setups larger than the example setup, it is planned to control the Production Cell [23] with it. It is already partially implemented, but the work is not completely finished yet.

Performing similar tests, as done in Section 2.3, really shows the advantages of using LUNA. Another planned test with the Production Cell is to control it with Arduinos [24]. The Production Cell has 6 separate production cell units (PCUs), each PCU is almost a separate part of the setup. Using one Arduino for each PCU seems like a nice experiment for distributed usage of LUNA. This would require support for a new platform within LUNA, which does not use operating system related functionalities.

References

[1] C.A.R. Hoare. *Communicating Sequential Processes*. Prentice-Hall, London, 1985.

[2] D.S. Jovanović, B. Orlic, G.K. Liet, and J.F. Broenink. gCSP: a graphical tool for designing CSP systems. In I. East, Jeremy Martin, P.H. Welch, David Duce, and Mark Green, editors, *Communicating Process Architectures 2004*, volume 62, pages 233–252, Amsterdam, September 2004. IOS press.

[3] B. Orlic and J.F. Broenink. Redesign of the C++ Communicating Threads library for embedded control systems. In F. Karelse, editor, *5th PROGRESS Symposium on Embedded Systems*, pages 141–156, Nieuwegein, NL, 2004. STW.

[4] H. Kopetz. *Real-Time Systems - Design Principles for Distributed Embedded Applications*. Kluwer Academic Publishers, 1997.

[5] J.F. Broenink, Y. Ni, and M.A. Groothuis. On model-driven design of robot software using co-simulation. In E. Menegatti, editor, *SIMPAR, Workshop on Simulation Technologies in the Robot Development Process*, November 2010.

[6] M.A. Groothuis, R.M.W. Frijns, J.P.M. Voeten, and J.F. Broenink. Concurrent design of embedded control software. In T. Margaria, J. Padberg, G. Taentzer, T. Levendovszky, L. Lengyel, G. Karsai, and C. Hardebolle, editors, *Proceedings of the 3rd International Workshop on Multi-Paradigm Modeling (MPM2009)*, volume 21 of *Electronic Communications of the EASST journal*. EASST, ECEASST, October 2009.

[7] N.C.C. Brown. C++CSP2: A Many-to-Many Threading Model for Multicore Architectures. In A.A. McEwan, W. Ifill, and P.H. Welch, editors, *Communicating Process Architectures 2007*, pages 183–205, July 2007.

[8] uClibc website, 2011. `http://www.uclibc.org/`.

[9] P.H. Welch, N.C.C. Brown, J. Moores, K. Chalmers, and B. Sputh. Integrating and Extending JCSP. In A.A. McEwan, W. Ifill, and P.H. Welch, editors, *Communicating Process Architectures 2007*, pages 349–369, July 2007.

[10] JamaicaVM website, 2011. `http://www.aicas.com/jamaica.html`.

[11] OROCOS website, 2011. `http://www.orocos.org/`.

[12] ROS website, 2011. `http://www.ros.org/`.

[13] E.W. Dijkstra. Notes on structured programming. In O.J. Dahl, E.W. Dijkstra, and C.A.R. Hoare, editors, *Structured programming*, chapter 1, pages 1–82. Academic Press Ltd., London, UK, 1972.

[14] D.S. Jovanović. *Designing dependable process-oriented software, a CSP approach*. PhD thesis, University of Twente, Enschede, The Netherlands, 2006.

[15] OpenWRT website, 2011. `http://www.openwrt.org/`.

[16] F. Fainelli. *The OpenWrt embedded development framework*. Free and Open source Software Developers' European Meeting (FOSDEM), January 2008.

[17] QNX website, 2011. `http://www.qnx.com`.

[18] Mordor website, 2011. `http://code.mozy.com/projects/mordor/`.

[19] M.M. Bezemer, M.A. Groothuis, and J.F. Broenink. Analysing gcsp models using runtime and model analysis algorithms. In P.H. Welch, H.W. Roebbers, J.F. Broenink, F.R.M. Barnes, C.G. Ritson, A.T. Sampson, D. Stiles, and B. Vinter, editors, *Communicating Process Architectures 2009*, volume 67, pages 67–88, November 2009.

[20] B. Veldhuijzen. Redesign of the CSP execution engine. MSc thesis 036CE2008, Control Engineering, University of Twente, February 2009.

[21] P.H. Welch and D.C. Wood. The Kent Retargetable occam Compiler. In Brian O'Neill, editor, *Parallel Processing Developments, Proceedings of WoTUG 19*, volume 47 of *Concurrent Systems Engineering*, pages 143–166, Amsterdam, The Netherlands, March 1996. World occam and Transputer User Group, IOS Press. ISBN: 90-5199-261-0.

[22] M. Charest and B. Stecher. Tick-tock - Understanding the Neutrino micro kernel's concept of time, Part II, April 2011. `http://www.qnx.com/developers/articles/article_826_2.html`.

[23] M.A. Groothuis and J.F. Broenink. HW/SW Design Space Exploration on the Production Cell Setup. In P.H. Welch, H.W. Roebbers, J.F. Broenink, and F.R.M. Barnes, editors, *Communicating Process Architectures 2009, Eindhoven, The Netherlands*, volume 67 of *Concurrent Systems Engineering Series*, pages 387–402, Amsterdam, November 2009. IOS Press.

[24] Arduino website, 2011. `http://www.arduino.cc/`.