Trustworthy Operating Systems

Eric Verhulst, Gjalt de Jong (Open License Society, Leuven, Belgium)
E-mail: eric.verhulst@OpenLicenseSociety.org , gjalt.dejong@OpenLicenseSociety.org,

OpenComRTOS:

An ultra-small network centric embedded RTOS designed to be correct with formal modeling

Summary

OpenComRTOS is one of the few Real-Time Operating Systems (RTOS) developed using formal modeling techniques. The goal of this project was to obtain a proven trustworthy component with a clean and high performance architecture useable on a wide range of embedded systems. The result is a scalable communication system with real-time capabilities. Besides a rigorous formal verification of the kernel algorithms, the resulting architecture has several properties that enhance the safety and real-time properties of the RTOS. The code size in particular is very small and typically 10 times less than a typical equivalent single processor RTOS.

1. Problem statement

Following a market research study for the European Space Agency in 2004, it was discovered that the majority of the RTOS (Real-Time Operating Systems) on the commercial as well as on open source market, cannot be verified or even certified, e.g. according to the DoD_178B or IEC61508 standards. This is due to a non-systematic software development approach, often bottom-up and with little documentation. This is remarkable as RTOS are widely used in embedded applications, often requiring properties of high reliability and safety. Similarly, software engineering is often done in a non-systematic way although well defined Systems Engineering Processes exist [3]. The software is rarely proven to be correct while formal model checkers exist. In the context of a unified systems engineering approach [4] we undertook a research project to follow a stricter methodology including formal model checking to obtain a network-centric RTOS as a trustworthy component.

2. General requirements for OpenComRTOS

The history for this project goes back to the early 1990's when a distributed real-time RTOS called Virtuoso (Eonic Systems) was developed for the INMOS transputer. This processor had build in support for concurrency and interprocess communication and was enabled for parallel processing by way of 4 communication links. Virtuoso allowed such a network of processors to be programmed in a topology transparent way. Later on the software evolved and was ported from single chip microcontrollers to systems with over a thousand Digital Signal Processors until the technology was acquired by Wind River and after a few years removed from the market.
The motivation for the OpenComRTOS project was to use the lessons acquired from 3 generations

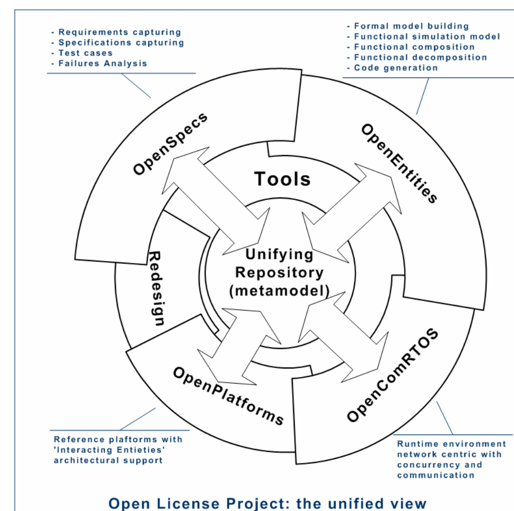

Figure 1 Open License SE methodology

of Virtuoso development. These lessons became

part of the requirements. We list the most important ones:

- **Scalability**: the RTOS should support from very small single processor systems to widely distributed processing systems interconnected through external networks like e.g. internet.
- **Network-centric.** The above scalability requirements imposes that data-communication is central in the architecture, but also that developing software is independent of the mapping onto the network topology.
- **Efficiency**: in multi-processing systems the essence is the communication. From the RTOS point of view the challenge is to keep the latency to a minimum while maximum performance is achieved when most of the critical code resides in the limited amount of on-chip fast memory.
- **Small code size**. This has a double benefit: performance and less complexity with potential sources of errors and side-effects.
- **Trustworthy**. As testing of distributed systems becomes very time consuming, it is mandatory that the system software can be trusted from the start. As errors typically occur in "corner cases", the use of formal methods was deemed necessary.
- **Maintainability and ease of development**. The code needs to be clear and simple and facilitate the development of e.g. drivers, the latter often been the weak point in system software.

In the context of the Systems Engineering methodology, the use of common semantics during all activities is crucial. Hence the final goal is to be able to generate most of the implementation code from the modeling and simulation phase. Considering the use of an "Interacting Entities" paradigm, this imposes the use of a runtime environment that supports concurrency and synchronization/communication in a native way between the concurrent Entities.

3. Initial architecture

While the above mentioned Virtuoso was a successful product, the goal was to improve on its weaknesses. The Virtuoso architecture was unique as it had two kernels inside. The lightweight nanokernel was mainly used for I/O and interprocessor communication while the microkernel provided priority based preemptive scheduling for user Tasks. This architecture was performant but very hard to port and maintain. Hence for OpenComRTOS a layered architecture was adopted but based on semantic layering. At the lowest level the functionality was limited to priority based preemptive multitasking with Tasks exchanging standardized Packets using an intermediate entity we called Ports. Hence, Tasks could synchronise and communicate using Packets and Ports. The Packets are the essential workhorse of the system. They have header and data fields

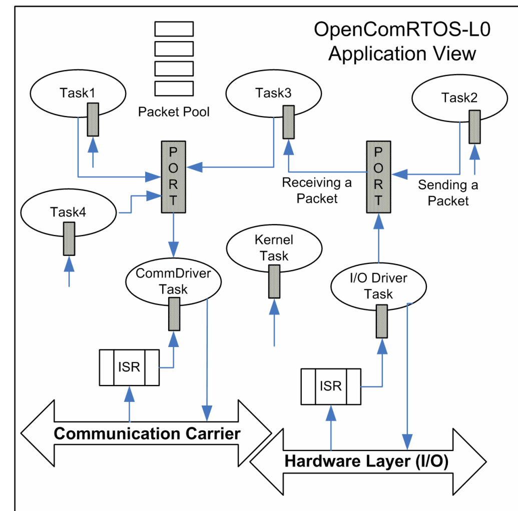

Figure 2 OpenComRTOS-L0 view

and are exclusively used for all services, rather than invoking e.g. function calls or using jump tables. Hence, it becomes straightforward to provide services that operate in a transparent way across processor boundaries. Packets are also very efficient as the kernel operation often comes down to shuffling around the packets (using handlers) between the system level datastructures.

At the next semantic level (L1) we wanted to add more traditional RTOS services like events, semaphores, queues, mailboxes, resources, etc. The concept was to achieve this using a second level of Packets whereby L0 Packets became full headers. This process is repeated for the next level L2 where we aim for support needed to address widely distributed nodes whereby the communication delay becomes substantial and the hard real-time behaviour becomes soft real-time. Such a level also requires support for mobility of code and of Entities. Finally, it was envisioned to keep the architecture simple and modular by developing the kernel as a Task as well as all drivers. All these Tasks have a 'Task input Port' for accepting Packets from other Tasks. This has some unusual consequences like the possibility to process

interrupts received on one processor on another processor, the kernel having a lower priority than the drivers or even having multiple kernel Tasks on a single node.

4. Systems (and Software) Engineering approach

The Systems Engineering approach from Open License Society is a classical one as defined in [4] but adapted to the needs of embedded software development. It is first of all an evolutionary process using continuous iterations. In such a process, much attention is paid to an incremental development requiring regular review meetings by several of the stakeholders. On the architectural level, the system or product under development is defined under the paradigm of "Interacting Entities", which maps very well on an RTOS based runtime system. Applied on the development of OpenComRTOS, the process was started by elaborating a first set of requirements and specifications. Next an initial architecture was defined. Starting from this point on, two groups started to work in parallel. The first group worked out an architectural model while a second group developed an initial formal model using TLA+/TLC [2]. This model was incrementally refined.

At each review meeting between the software engineers and the formal modeling engineer, more details were added to the model, the model was checked for correctness and a new iteration started. This process was stopped when the formal model was deemed close enough to the implementation architecture. Next, a simulation model was developed on a PC (using Windows NT as a virtual target). This code was then ported to a real 16bit microcontroller [5]. On this target a few target specific optimizations were performed on the implementation, while fully maintaining the design and architecture. The software was written in ANSI C and verified with a MISRA rule checker. [8]

5. Lessons from using formal modeling

The initial goal of using formal techniques was to be able to prove that the software is correct. This is an often heard statement from the formal techniques community. A first surprise was that each model gave no errors when verified by the TLC model checker. This is actually due to the iterative nature of the model development process and partly its strength. From an initial rather abstract model, successive models are developed by checking them using the model checker and hence each model is correct when the model checker finds no illegal states. As such, model checkers can't proof that the software is correct. They can only proof that the formal model is correct. For a complete proof of the software the whole programming chain should be verified as well as the target hardware be modeled and verified as well. This is an unachievable result due to its complexity and the resulting state space explosion. It was nevertheless attempted in the Verisoft [6] project. The model itself would be many times larger than the software being developed. It indicates however that if we would make use of verified target processors and verified programming language compilers, the model checker becomes practical as limited to modeling the application.

Other issues were discovered in relation to the use of formal modeling. A first issue is that the TLC model checker declares every action as a critical section, whereas e.g. in the case of a RTOS, many components operate concurrently and real-time performance dictates that on a real target the critical sections are kept as short as possible. While this dictates the avoidance of shared data structures, it would be helpful to have formal model assistance that indicates the required critical sections.

The final issue is the well known problem of state space explosion. Just modeling a small OpenComRTOS application the TLC model checkers has to examine millions of states, exponentially taking more time for every Task added to the model. This also requires increasing amounts of memory and limits the model checking to subsets of the whole architecture.

6. Benefits obtained from using formal modeling

As was outlined above, the use of formal modeling was found to result in a much better architecture. This benefit is the result of the process of successive iteration and review, but also because formal models checkers provide a level of abstraction away from the implementation. In the project e.g. we found that the semantics associated with specific programming terms involuntarily influence choices made by the architecting engineer. An example was the use of both a waiting list and a buffer for a Port, which is one of the main concepts of OpenComRTOS. A waiting list is associated just with waiting but one overlooks that it also provides buffering behavior. Hence, one waiting list is sufficient resulting in a smaller and cleaner architecture. The formal

modeling and abstract level has helped to introduce, define and maintain orthogonal concepts in the architecture. Orthogonality is the key to have small and safe, i.e. reliable, designs. Similarly, even if there was a short learning curve to master the mathematical notation in TLA, with hindsight this was an advantage vs. e.g. using SPIN [7] that uses a C-like syntax. The latter leads automatically to thinking in terms of an implementation code with all its details whereas the abstraction of TLA helped to think in more abstract terms. This also highlights the imPortance of specifying first before implementation is started.

7. Novelties in the architecture

OpenComRTOS has a semantically layered architecture. At the lowest level (L0) the minimum set of Entities provides everything that is needed to build a small networked real-time application.

The Entities needed are **Tasks** (having a private function and workspace), an Interaction Entity we called an L0_Port to synchronize and communicate between the Tasks. **Ports** act like channels in the tradition of Hoare's CSP but allow multiple waiters and asynchronous communication. One of the Tasks is a kernel Task scheduling the Tasks in order of priority and managing and providing Port based services. Driver Tasks handle inter-node communication. Pre-allocated as well as dynamically allocated Packets are used as a carrier for all activities in the RTOS such as: service requests to the kernel, Port synchronization, data-communication, etc. Each Packet has a fixed size header and data payload with a user defined but global data size. This significantly simplifies the management of the Packets, in particular at the communication layer. A router function also transparently forwards Packets in order of priority between the nodes in a network.

OpenComRTOS L0 therefore is a distributed, scalable and network-centric operating systems consisting of a packet-switching communication layer with a scheduler and Port-based synchronization. This architecture has proven to be very efficient. E.g. a minimum single processor kernel can have a code size of less than 1 Kbyte, with 2 Kbytes for the multi-processor version.

In the next semantic level (L1) services and Entities were added as found in most RTOS: Boolean events, counting semaphores, FIFO

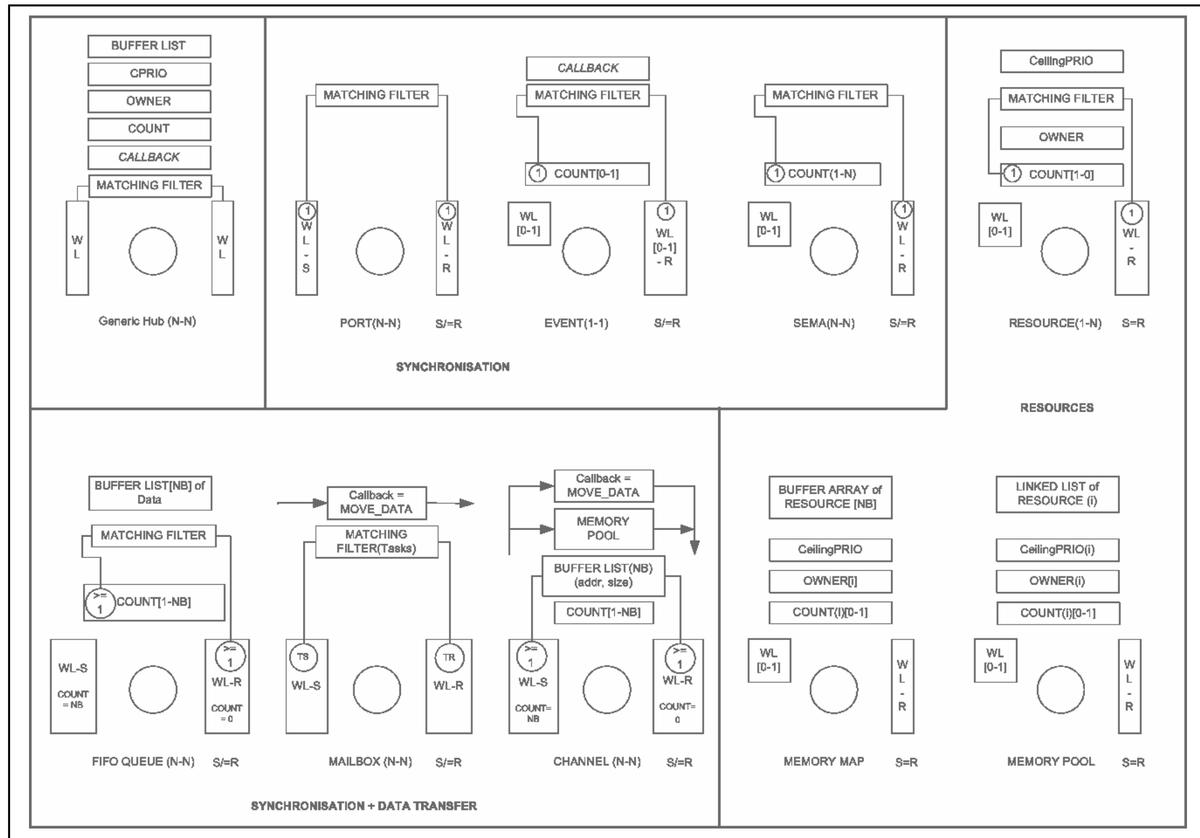

Figure 3. L1 services derived from a generic "Hub"

queues, resources, memory pools, mailboxes, etc. The formal modeling has allowed defining all such Entities as semantic variants of a common and generic entity type. We called this generic entity a "Hub". In addition, the formal modeling also helped to define "clean" semantics for such services whereas ad-hoc implementations often have side-effects. In Table 1 we summarise the semantics.

The services are also offered in a non-blocking variant (_NW), a blocking variant (_W), a blocking with timeout variant (_WT) and an asynchronous variant when this makes sense. All services are transparent for the topology and the network mapping of Task and kernel Entities onto this network. See Tables 1 and 2 for details on the semantics.

L1 Entity	Semantics
Event	Synchronisation on Boolean value. Waiting list on both sides.
Counting Semaphore	Synchronisation with counter allowing asynchronous signaling.
Port	Synchronisation with exchange of a Packet.
FIFO queue	Buffered communication of Packets. Synchronisation when queue is full or empty.
Resource	Event used to create a logical critical section. Resource have an owner Task when locked
Critical Section	Entity creating a global critical section based on locking multiple resources.
Memory Pool	Linked list of memory blocks protected with a resource
Mailbox	Synchronising entity with matching filter on Task ID. Communication happens as side-effect.
Channel	Asynchronous communication between Tasks with buffering using memory pools. Communication as a side-effect.

Table 1. Semantics of L1 Entities.

As the use of a single generic entity allowed a much greater reuse of code, the resulting code size is at least 10 times less than for an RTOS with a more traditional architecture. One could of course remove all such application-oriented services and just use the Hub based services. This has however the drawback that the services loose their specific semantic richness. E.g. resource locking clearly expresses that the Task enters a critical section in competition with other Tasks. Also erroneous runtime conditions like raising an event twice (with loss of the previous event) are easier to detect at the application level than when using a generic Hub.

In the course of the formal modeling we also discovered weaknesses in the traditional way priority inheritance is implemented in most RTOS and we found a way to reduce the total blocking time. In single processor RTOS systems, this is less of an issue but in multi-processor systems, all nodes can originate service requests and resource locking is a distributed service. Hence the waiting lists can grow much longer and lower priority Tasks can block higher priority ones while waiting for the resource. This was solved by postponing the resource assignment till the rescheduling moment.

Services variants	Synchronising Behaviour
"Single-phase" services	
_NW	Non Waiting: when the matching filter fails the Task returns with a RC_Failed
_W	Waiting: when the matching filter fails the Task waits until such events happens.
_WT	Waiting with a time-out. Waiting is limited in time defined by the time-out value.
"Two-phase" services	
_Async	Asynchronous: when the entity is compatible with it, the Task continues independently of success or failure and will resynchronize later on. This class of services is called "two-phase" services.

Table 2. Service synchronization variants

Finally, by generalization, also memory allocation has been approached like a resource locking service. In combination with the Packet Pool, this opens new possibilities for a safe and secure management of memory. E.g. the OpenComRTOS architecture is free from buffer overflow by design.

For the third semantic layer (L2), we will add dynamic support like mobility of code and of kernel Entities. A potential candidate is a light weight virtual machine supporting capabilities as modeled in pi-calculus. [9] This is the subject of further investigations and will be reported in subsequent papers.

8. Inherent safety support

By its architecture the L0 and L1 semantic layers are all staticly linked, hence an application specific image will be generated by the compiler tools. As we don't consider security risks for the moment, our concern is limited to verifying if the code is inherently safe.

A first level of safety is provided by the formal modeling approach. Each service is intensively modeled and verified with most "corner cases" detected during design time prior to writing the code. A second level is provided by the kernel services. All services have well defined semantics. Even when asynchronously used, the services become synchronous when available resources become depleted. At such moment a Task becomes waiting allowing other Tasks to proceed and free up resources (like Packets, space in the buffers, etc.). Hence, the systems becomes "self-throttling". A third level is provided by the data structures, mostly based on Packets. All single-phase services uses statically allocated Packets that are part of the Task context. These Packets are used for service requests, even when going across processor boundaries. They also carry the return values. For two phase services Packets must be allocated from a Packet Pool. When the Pool is empty, the system will start to throttle until Packets are released. Another specific feature of the architecture is that buffers cannot overflow. In the worst case the application programmer will not have defined enough Packets in the Pool and buffers will stop growing when all Packets are in use. A last level is the programming environment. All Entities (at L0 and L1) are defined statictly so they are generated together with all other system level datastructures by a tool and hence no Entities can be created at runtime. Of course, dynamic support at the L2 level will require extra support. However this can only be achieved reliably with hardware support e.g. to provide protected memory spaces. The same applies to the use of the stack spaces. In OpenComRTOS interrupts are handled on a private and separate stack so that the Task's stack spaces are not affected. On the MLX16 such a space can be protected but it is clear that such inexpensive mechanism should be a must on all embedded processors for all stack spaces. A full MMU is not only too complex and too large but it is not even needed. The kernel also has various threshold detectors and provides support for profiling, but the details are outside the scope of this paper.

8. Measurements on real execution targets

We shortly summarize the results obtained. Although fully written in ANSI-C (except for the Task context switch), the kernel could be reduced to less than 1 Kbytes single processor and 2 Kbytes with multi-processor support (measured on a 16bit Melexis microcontroller). A sample application with two Tasks and two Ports required just 1230 bytes of program memory and 226 bytes of data memory (static and dynamic).

When adding L1 services (events, semaphores, resources and FIFO queues) the code increased with less than 1 kBytes. An overview in given in Table 1. All figures are for non hand optimised code written in ANSI C compiled on the MLX16 16bit microntroller. The reader will recognize first of all that the architecture has very little penalty for providing network enabled services.

A second port was undertaken to the Windows NT platform, serving as a simulator as well as host node. Further Ports are underway to MicroBlaze, SPARC and the Cell processor.

OpenComRTOS L1 code size figures				
	MP FULL		SP SMALL	
	L0	L1	L0	L1
L0 Port	162		132	
Hub shared		574		400
L1 Port		4		4
Event		68		70
Sema-phore		54		54
Resource		104		104
FIFO		232		232
Resource List		184		184
Total L1 services		1220		1048
Grand Total	**3150**	**4532**	**996**	**2104**

Table 3. Code size figures L0 and L1

MP Full: with router, no driver Tasks
SP Small: single processor, no router
All services: (_W,_WT,_NW,_Async)

On the MLX16 microcontroller a processor specific version was developed. This version is limited to 16 Tasks and uses some processor

specific instructions to reduce the overhead. Such minimal L0 RTOS could be made to fit in 904 bytes, whereas adding L1 Events, Ports and Resources only added 240 bytes to the code. Although this microcontroller runs from flash with no cache at about 6.5 Mips, the interrupt latency from a timer interrupt to the first instruction in a Task where the timer register can be read is only 52 microseconds. For a round-trip loop between two Tasks sending and receiving Packets using 2 Ports, we measured 10740 loops/second. This is about 93 microseconds for two Task switches, two L0_SendPacket and two L0_ReceivePacket services. All code, including the RTOS was compiled using the GCC compiler. The microcontroller has 4 registers.

A second version was ported on top of Windows NT and using sockets to simulate internode communication. The same test application as the one on the MLX16 could be generated for this "virtual" target by recompiling the source code and linking with the target specific libraries. This demo was transparently distributed over a number of PCs connected over a LAN. Code size figures and performance times are not really relevant for this target, but it demonstrates how a widely distributed and heterogenous network can be supported. E.g. using this scheme an MLX16 node can read a sensor and transmit it over an UART to a "host PC" running an instance of OpenComRTOS. This PC then communicates with another PC over a VPN whereby an operator sends a control command back to the MLX16.

10. Impact on software quality

RTOS kernel code is typically known to be "black art" programming. This is due to the concurrent and asynchronous nature of the software, direct interfacing to the hardware, context switching and the requirement to produce not only performant but also compact code. Hence we were curious to see if the formal development path we followed would have an impact on the code quality. Therefore the code was subjected to the MISRA tool checker and quality software metrics of LDRA. The MISRA standard is a set of 140 rules that a program written in C should adhere to be safe. Most of these rules prevent programmers from using all the 'dirty' tricks, most often with side-effects that C allows. The source code had no problems passing this check. Also the quality of the source was very high according to the metrics generated by the LDRA tools. When the score was lower, it was due to the presence of too many comment lines or when the source file contained some in-line assembler like the context switch. The conclusion is that small optimized code doesn't need to be hand crafted provided a lot of thought went into defining a clean architecture. The major gain is hence achieved by defining a globally optimized architecture and less from looking for punctual optimizations.

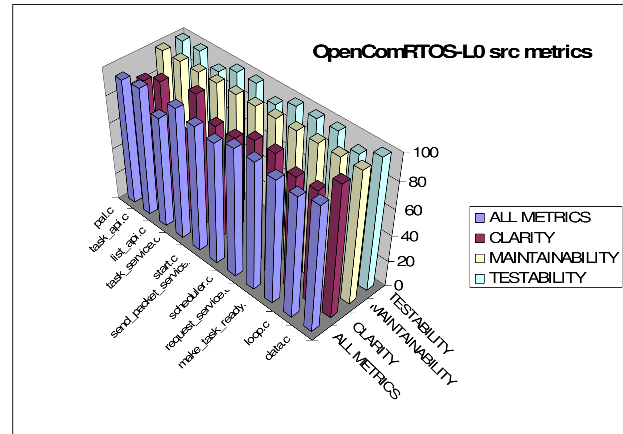

Figure 4. Quality metrics by LDRA**.**

11. Future developments and research

Above we already identified the need for the model checkers to detect the minimal critical sections. Another area of research is how to maintain consistency between the formal model and the implementation. This will require that the formal model can be used as a reference and requires that the source is generated rather than written by the software engineer.

Future OpenComRTOS developments will focus on adding more safety and security properties to a SW/HW co-design pair of OpenComRTOS and processor. Formal modeling should contribute in identifying minimum architectures that still are providing safety and security in the resource constrained domain of deeply embedded systems.

Another area of interest is to find a better way to separate orthogonally the priority based scheduling from the logical behavior of the kernel Entities. E.g. the use of priority inheritance support results in this code being mixed up in the manipulation of the data structures (e.g. to sort waiting lists). This makes the code more convoluted to read and understand while the impact is only on the timely behavior of the application.

10. Conclusion

The OpenComRTOS project has shown that even for software domains often associated with 'black art' programming, formal modeling works very well. The resulting software is not only very robust and maintainable but also very performing in size and timings and inherently safer than standard implementation architectures. Its use however must be integrated with a global systems engineering approach as the process of incremental development and modeling is as important as using the formal model checker itself. The use of formal modeling has resulted in many improvements of the RTOS properties.

Acknowledgements

The OpenComRTOS project is partly funded under an IWT project for the Flemish Government in Belgium. The formal modeling activities were provided by the University of Gent.

References

1. OpenComRTOS architectural design document on www.OpenLicenseSociety.org
2. TLA+/TLC home page on http://research.microsoft.com/users/lamPort/tla/tla.html
3. INCOSE www.incose.org
4. Open License Society www.OpenLicenseSociety.org
5. www.Melexis.com
6. www.verisoft.de
7. www.spin.org
8. www.misra.org
9. Robin Milner. Communicating and Mobile Systems: the Pi-Calculus. Cambridge University Press 1999.