OpenComRTOS – Distributed RTOS development using formal modeling methods

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

Abstract. 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. These goals were achieved. 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. In the project we used the TLA/TLC formal modeling toolset of Leslie Lamport.

Keywords. RTOS, Formal Methods, Trustworthy, Safety, Security, Network centric

1 Introduction

Following a market research study for the European Space Agency in 2004, it was discovered that the majority of the RTOS on the commercial as well as open source market, cannot be verified or even certified. 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 seldom proven to be correct while formal model checkers exist. In the context of Open License Society's 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 Systems (and Software) Engineering approach

The Systems Engineering approach adopted by Open License Society is a classical one as defined in [3] but adapted to the needs of embedded software development. It is first of all an evolutionary process using constant iteration reviews. In such a process, much attention is paid to an incremental development requiring often review meetings by several of the stakeholders. On the architectural level, the system or product under developed 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 (led by Prof. Boute of the University in Gent) 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 virtual target). This code was then ported to a real 16bit microcontroller [5] and optimized. On this target target specific optimizations were implemented as well. The software was written in ANSI C and verified with a MISRA rule checker.

3 Application of formal modeling

3.1 Lessons learnt

The initial goal of using formal techniques was to be able to proof 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 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 datastructures, it would be helpful to have formal modelers that indicate the real 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 a few million states, exponentially taking more time for every task added to the model. This also requires increasing amounts of memory.

3.2 Benefits obtained

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 terms used when programming involuntarily influence choices made by the architecting engineer. E.g. a waiting list is associated just with waiting but one overlooks that it also provides buffering behavior. 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.

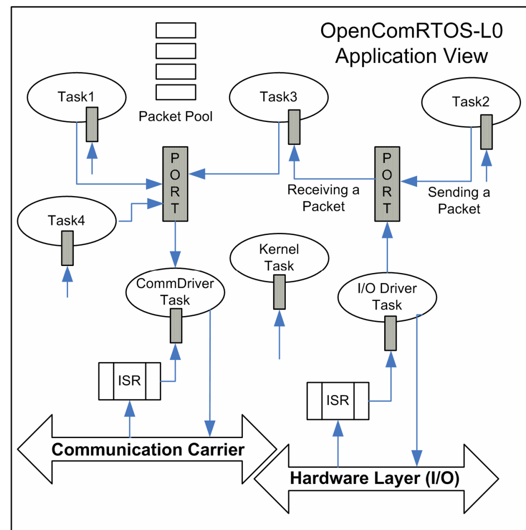
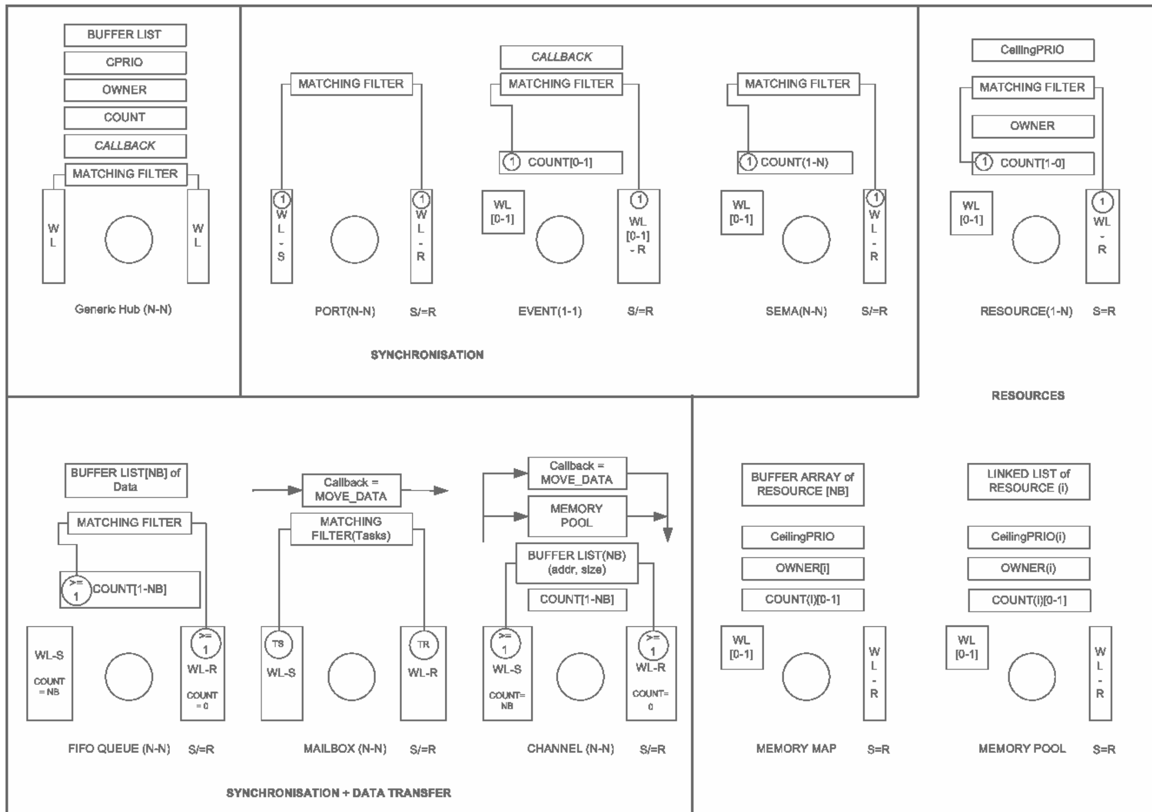


Figure 1 OpenComRTOS-L0 view

4 Results

4.1 Architecture novelties

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 synchronise 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 simplifies a lot 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. 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 queues, resources, memory pools, mailboxes, etc. The formal modeling has allowed defining all such entities as semantic variants of a common more generic entity type. We called this generic entity a “Hub”.

As this allows 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 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.

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 are looking at adding dynamic support like movable code and kernel entities. A potential candidate is a light weight virtual machine supporting capabilities as modeled in pi-calculus. This is the subject of further investigations.

4.2 Performance Results

Although fully written in ANSI-C (except e.g. the 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 successful test application with two tasks and one Port required just 1230 bytes of program memory and 226 bytes of data memory (static and dynamic). For adding L1 services (events, semaphores, resources and FIFO queues) the code increased with less than 1 Kbytes. A second port was undertaken to the Windows NT platform, serving as a simulator as well as host node. Further ports are underway.

5 Conclusions

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 a standard implementation architecture. 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 are provided by the University of Gent. Melexis is co-sponsoring the effort by providing the Melexis microcontroller as a resource constrained target for use in embedded automotive electronics.

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