An industrial case: pitfalls and benefits of applying formal methods to the development of a network-centric RTOS

Eric Verhulst, Gjalt de Jong, Vitaliy Mezhuyev
Open License Society, Leuven, Belgium
E-mail: {eric.verhulst,gjalt.dejong, vitaliy.mezhuyev}@OpenLicenseSociety.org

Abstract. This paper describes a project to develop a network-centric RTOS from scratch using formal methods. The (initial) purposes of the project was to get acquainted with the use of formal methods for software engineering and to obtain a trustworthy RTOS as a component for building networked embedded systems. The work was done by a small, distributed team that had no prior experience on using formal methods and with a small budget. The outcome is that the use of formal methods is most useful as an architectural design method, more than as a formal verification of software code. The resulting software has many properties that were not anticipated at the beginning and would likely not have been achieved without the use of Formal Methods

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

Acknowledgements

The OpenComRTOS project was partly funded under an IWT project for the Flemish Government in Belgium. The formal modeling activities are provided by the University of Gent. Melexis was industrial sponsor.

1 Problem statement

Real-Time Operating Systems are an essential component in most embedded systems. They are essential when the application becomes complex and safety critical. They provide a way to organize the application in a set of modules that interact, the scheduler helps in achieving predictable real-time behavior, and they allow the application to recover from run-time error conditions.

Nevertheless, almost none of the commercial and open source RTOS-es have been certified according to standards like IEC61508 or DO178. Almost none have been formally verified. Part of the reason is historical: RTOSes are fairly complex and highly concurrent pieces of software that in addition must provide good performance with as little as possible resources. Hence, RTOSes are often developed by very skilled software engineers, but often following a bottom-up approach with little documentation, preventing even certification.

Open License Society undertook the OpenComRTOS project in 2004 with the aim to develop a novel network-centric RTOS. Formal methods were used from the start with much effort going into finding the right architecture and being able to verify that the software is correct.

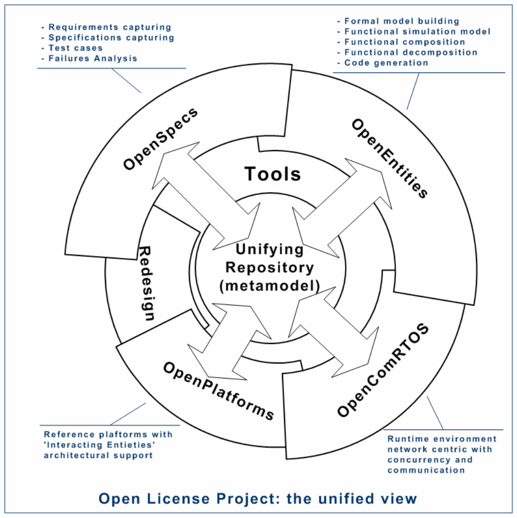

- Requirements capturing
- Specifications capturing
- Test cases
- Failures Analysis
- Formal model building
- Functional simulation model
- Functional composition
- Functional decomposition
- Code generation
OpenSpecs
OpenEntities
Tools
Unifying Repository (metamodel)
Redesign
OpenPlatforms
OpenComRTOS
Reference plaftorms with 'Interacting Entieties' architectural support
Runtime environment network centric with concurrency and communication
Open License Project: the unified view

Figure 1 Open License SE methodology

We also noted related work by Iain D. Craig [11][12] when this project was finished. This work is however rather different. It is mainly concerned with the formal specification and refinement of existing Operating Systems. The author shows that this is viable. Our work has indicated that formal methods provide serious benefits as well when used for designing new architectures from the very beginning, ven for nontrivial pices of software like RTOS.. As a result, formal verification of the final architecture is also a lot more straightforward because it results in a much cleaner architecture.

2 Systems (and Software) Engineering approach

The Systems Engineering approach developed by Open License Society is a classical one as defined in [4] but adapted to the needs of embedded software development. It is an evolutionary iterative process. 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. When programming with RTOS, the appliction is split over number of concurrent entities called "Tasks", scheduled in time by the RTOS scheduler. The "interact" through RTOS services, essentially points of synchronization but with a service specific semantic behaviour. In OpenComRTOS these services decouple the tasks completely from each other. 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] Finally the reverse process was undertaken. For each service class a formal model was built matching the implementation and essential properties were verified.

3 Lessons from using formal modeling

3.1. Selecting a methodology

Formal techniques basically fall into two categories. First we have model checkers: a model of the software is constructed at an abstract level and the model checker will basically verify that specified properties are never violated and if they are a trace of a counter-example will be provided. A second class of formal techniques are so-called proofing systems. They allow to proof by deduction and aided by a computing machine that a certain property holds.
Given that the project started with a clean slate and the strong architectural nature of the project we opted to use a model checker. It must be noted however that model checkers, neither proof systems allow to verify just any set of properties. E.g. most model checkers are only suitable for verifying event triggered systems as e.g. numerical properties quickly result in a state space explosion, restricting their use to rather small systems. Fortunately for an RTOS, this is less of an issue.
A first observation is that while there are many tools and methods available, most of them are based on the same principles. However, many of the tools we found are academic and suffer from lack of robustness, performance or ease of use, clearly indicating that this is still an emerging discipline. Also when used by commercial vendors, the formal tools are often hidden and do their work in the background. This obliterates the need to be mathematical proficient and user can stay in the problem domain, instead of the math solution domain, but no such integration was found that applied to our project.
While we had an initial bias toward using SPIN [7], in the end it was decided to use TLA/TLC from Leslie Lamport. [2] Although the mathematical notation of the TLA language was first considered a hindrance versus the C-like Promela language of SPIN, in the end this has proven to be a major benefit as it forced to reason in a much more abstract way about the RTOS.

3.2. Why are there no errors?

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 was 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 as well as the target hardware should be modeled and verified as well. In the ideal case, the software should even be generated from the formal models. This is today an unachievable result due to its complexity and the resulting state space explosion. 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. E.g. 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.

Nevertheless, a major benefit of using the model checker has proven to be its abstraction. The models developed first in/in the beginning of the project had to be discarded after it was clear that they reflected how a programmer would write the software, often by unconsciously taken implementation decisions, resulting in unnecessary complexity. Once this was understood, (re)developing the models was much more straightforward.

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

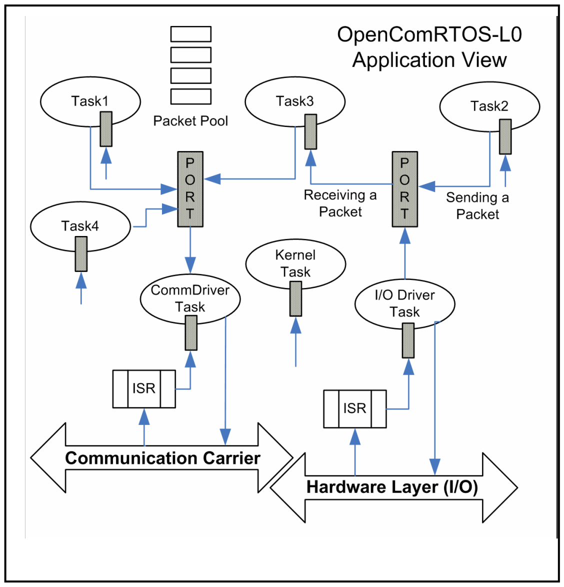


Figure 2 OpenComRTOS-L0 view

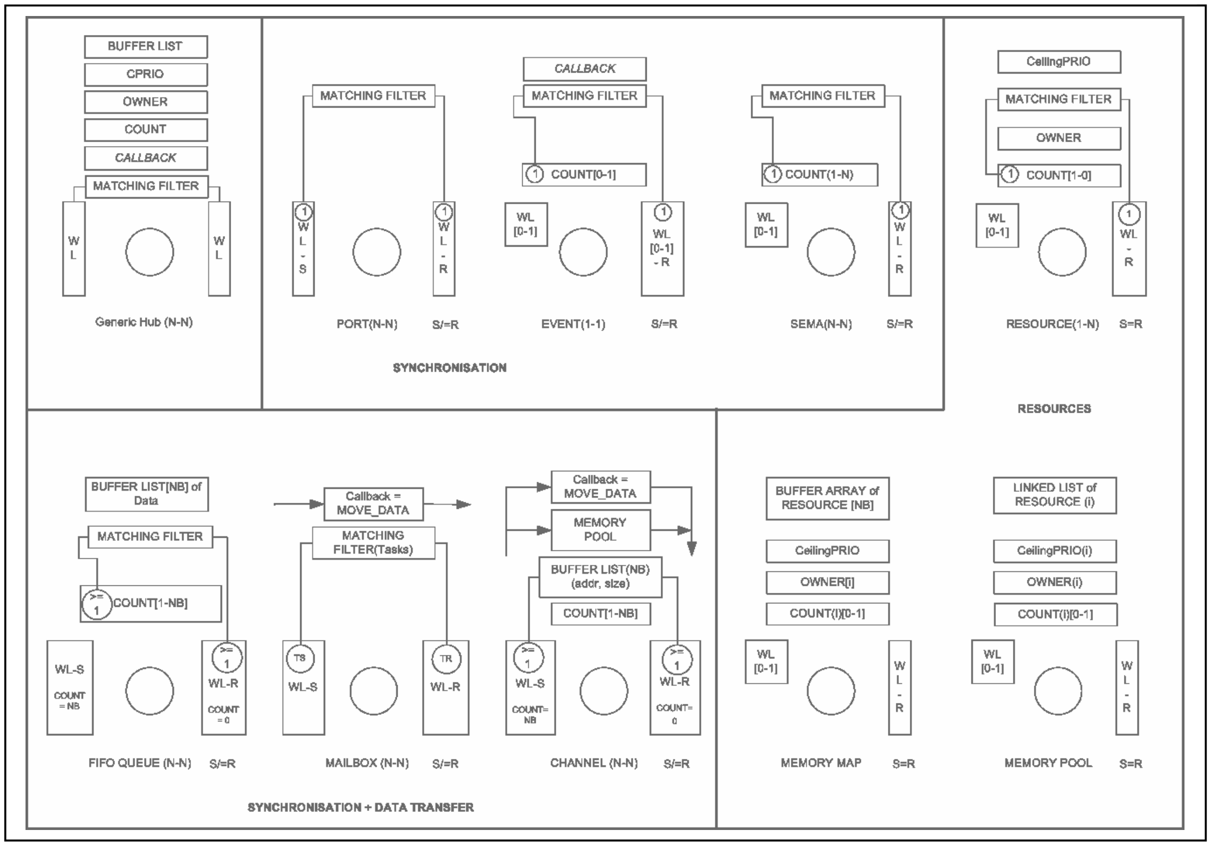
amounts of memory and limits the model checking to subsets of the whole architecture. However, this was not a real issue as the architecture is generic and based on a message passing protocol that is independent of the size of the system. The algorithmic logic of the RTOS kernel also makes no difference between local or remote services, making it independent of the topology of the target network and hence there was no need to make the network topology explicit.

4 A thin boundary between past experience, creativity and model checking

For completeness, we need to mention that some of the elements of the OpenComRTOS architecture were inherited from a previous distributed RTOS (Virtuoso [4]) that was developed in a traditional way, and with some inspiration from CSP. The communication layer of this distributed RTOS used packets but the kernel was a large jump table. We had also experienced issues with portability and scalability. Finally, the third generation of the Virtuoso RTOS was loosing performance through what we can call "feature bloating". Nevertheless, it was difficult to see how a better architecture could be found that would at the same time provide improvements in terms of code size, safety, security and scalability properties. In addition we defined as objective that it should be able to run from memory restricted multi-core CPUs to widely distributed processing nodes running legacy software.

Formal modeling has helped a lot in formalizing the problem and as a result we can claim success beyond initial expectations.

Fig 4. L1 RTOS Interaction entities based on a generic Hub.



5 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 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.

As the use of a single generic entity allowed a much greater reuse of code, the resulting code size is about 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.

An unexpected side-effect of the use of Hub entities, is that the set of services can be expanded independently of the kernel itself. A Hub is a generic synchronization entity and the Hub semantics are determined by the synchronization predicate and by the predicate function following successful synchronization. The result is not only that the RTOS can be made application specific, it also provides better performance and more safety as most of the services and the driver code execute in the application domain, leaving the essential RTOS functions to a small kernel function.

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.

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.

6 Results obtained 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 one Port required just 1230 bytes of program memory and 226 bytes of data memory (static and dynamic). More information is available in [4]

7 Formal verification

This project would have been incomplete if we had not attempted a formal verification of the source code. In the end this proved to be quite straightforward because the orthogonal and clean architecture allowed to check each service using a similar pattern. Following issues however must be mentioned:

- We did not find tools and methods that allowed to verify our asynchronous and concurrent design (inevitable for a RTOS) at the source code level. Tools only exist for static and synchronous programs [9][10]
- It was practically impossible but also unnecessary to verify the kernel as a whole. Hence we verified the algorithms for each service class independently. Given the orthogonality of protocol based architecture (by using packets), this is sufficient.
- The hardest part remained to find all properties to check for. A lot of these properties look rather trivial at first sight and our human brain has a tendency to overlook them.
- The final issue is related to the programming in C itself. It is clear that this language is a major source of errors. Hence, some errors were found at the programming level that no formal verification would ever find.
- However, the fact that the formal modeling helped a lot in achieving such a clean and orthogonal architecture, verification as well as at the abstract level by using a formal model checker as well as at the language level was a lot easier, because the complexity is minimized and the code size is much smaller than comparable hand written code.

8 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.

8 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.

Formal modeling and formal verification have proven to be very powerful engineering tools and hence it can not be emphasized enough how many problems in the software world can be avoided by a systematic use from the very beginning.

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. Bruno Blanchet, Patrick Cousot, Radhia Cousot, Jérôme Feret, Laurent Mauborgne, Antoine Miné, David Monniaux & Xavier Rival. Design and Implementation of a Special-Purpose Static Program Analyzer for Safety-Critical Real-Time Embedded Software, invited chapter. In *The Essence of Computation: Complexity, Analysis, Transformation. Essays Dedicated to Neil D. Jones*, T. Mogensen and D.A. Schmidt and I.H. Sudborough (Editors). Lecture Notes in Computer Science 2566, pp. 85—108, Springer. . http://www.astree.ens.fr/
10. Clarke, Edmund and Kroening, Daniel and Lerda, Flavio, A Tool for Checking {ANSI-C} Programs, Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2004), Springer http://www.cprover.org/cbmc/