Eric Verhulst · Raymond T. Boute
José Miguel Sampaio Faria
Bernhard H.C. Sputh · Vitaliy Mezhuyev

Formal Development of a Network-Centric RTOS

Software Engineering for Reliable Embedded Systems

Springer

Formal Development of a Network-Centric RTOS

Eric Verhulst • Raymond T. Boute
José Miguel Sampaio Faria • Bernhard H.C. Sputh
Vitaliy Mezhuyev

Formal Development of a Network-Centric RTOS

Software Engineering for Reliable Embedded Systems

Springer

Eric Verhulst
Altreonic NV
Gemeentestraat 61AB1
B3210 Leuven, Belgium
Eric.Verhulst@lancelot.be

José Miguel Sampaio Faria
Rua Sra das Boas Novas 776
4935-490 Mazarefes
Portugal
jmfaria@criticalsoftware.com

Vitaliy Mezhuyev
Open License Society
Zavelstraat 160
3010 Leuven
Belgium
Vitaliy.Mezhuyev@openlicensesociety.org

Raymond T. Boute
Department of Information Technology
Universiteit Gent
Faculty of Engineering
St. Pietersnieuwstraat 41
9000 Gent
Belgium
boute@intec.UGent.be

Bernhard H.C. Sputh
Open License Society
Zavelstraat 160
3010 Leuven
Belgium
bernhard.sputh@openlicensesociety.org

ISBN 978-1-4419-9735-7 e-ISBN 978-1-4419-9736-4
DOI 10.1007/978-1-4419-9736-4
Springer New York Dordrecht Heidelberg London

Library of Congress Control Number: 2011933844

Preface

How can one improve with a factor of 10 on something that has already the reputation of being highly optimised? The answer lies in ignoring the most often wrong assumption that it is already highly optimised and by going back to basics. This inevitably includes developing a new formalisation of the problem at hand. In our case, this meant thinking anew about what a distributed RTOS (Real Time Operating System) is all about. What is the core functionality of an RTOS, of a distributed RTOS? Is there a clean way to handle task synchronisation and communication? The result was the unique network-centric OpenComRTOS project described in this book.

Taking this as an opportunity, we wanted to use formal methods to prove the final implementation. It turned out that formal methods can help to prove an implementation, but they really shine when used to model the architecture at an abstract level before any implementation is done. Their use has shown us again how much we are all influenced by what we know. After all our brains have a hard time reasoning without prior knowledge. Hence, our brains tend to look for known patterns so that known rules can be applied.

Looking for better and new solutions is hampered by prior knowledge. Formal methods help us because they allow us (or some would say: force us) to think at a more abstract level, our vision being less cluttered by implementation details. The result obtained in the project was a very clean and scalable architecture while verification had almost become trivial. There is also a general assumption that trustworthy means complex and large. Great was the surprise, however, when we discovered it resulted in the opposite. The RTOS was measured to be up to 10 times smaller than a previously hand coded version that had been tweaked over several years and used in demanding systems. This means less resources and less power are needed. So, to make the world less energy-hungry, use formal methods.

This project has to some extent reinvented the very concept of what an RTOS is. It is a way to model, it is a way to simulate, it is a way to verify, it is a way to program in a scalable and portable way concurrent systems. But our quest does not stop.

OpenComRTOS is also an enabler for new functionality that is still being researched while the book is being written. A lot of the work has to do with researching the correct semantics to support e.g. composability, dynamic resource scheduling and fault tolerance. Ultimately, it might result in new hardware.

Last but not least, formal methods have proven not to be so hard to use as it was assumed to be. The project also demonstrated the strength of team work. Communication in a well working team is ultimately the way to get rid of the assumptions our brains involuntary make. Formal methods again help by replacing intuition by abstraction. This book is not an academic one. It describes aspects that were explored during a real industrial project to develop a distributed RTOS from scratch using formal methods. Therefore it contains as well a broad discussion on the context in which such RTOS are used, as well as deep technical details of some of the formal models used. But as such, the description is not complete because it describes a project, not a theory.

The book is organised as follows: In the first two chapters, we sketch the domain of interest: trustworthy embedded real-time distributed systems. We discuss the challenges to develop applications and systems in this domain and why formal methods are becoming essential tools for the engineer working in this field. We derive from it the requirements and specifications for OpenComRTOS. In the following two chapters we look at what formal methods and tools are available and introduce TLA+/TLC that was finally selected and used in the project. Subsequently, we discuss the formal TLA+ models, as well as the architecture, of OpenComRTOS. We dwell a bit deeper on the interaction semantics and provide an overview of the code size and performance results. For the interested user the appendix includes a usage tutorial, as well as the mathematical and logic foundations behind temporal logics like TLA+. The appendix also contains the TLA+ and SPIN models used to compare both formalisms in Chap. 3.

For the interested reader, a free version of OpenComRTOS for PC is available from www.altreonic.com. This version also acts as a simulator and cross development environment for multi-node targets.

Acknowledgements

This work has been made possible by the support of many people and organisations:

- Alexander Keda for developing the verification models and code generators.
- Anatoliy Konovalenko for developing the RTOS unit tests.
- Andrey Nitsenko for developing the graphical event tracer.
- Annie Dejonghe for moral support and administrative support.
- Bernhard Sputh for managing the release of the product and porting the RTOS.
- Dimitry Panfilov for developing the first visual front-end and porting the RTOS.
- Gjalt De Jongh for his conceptual discussions and first implementations.
- José Miguel Faria for developing the first formal models.

- Raymond Boute for his deep knowledge of formal techniques.
- Vitaliy Mezhuyev for his meta-modelling input.

The project was also financially supported by IWT of the Flemish Region and Melexis NV. Melexis also provided the first target processor.

Linden Eric Verhulst

Contents

List of Figures

List of Tables

Part I
Trustworthy Embedded Systems

Chapter 1
Introduction: OpenComRTOS Role in a Unified Systems Engineering Methodology

OpenComRTOS is part of a systematic, formalised systems and software engineering methodology for embedded systems with a supporting environment and tools. While OpenComRTOS can be used independently of it, users will benefit from using the methodology in an integrated way. This methodology is characterised by two key concepts: unified semantics and interacting entities. When used in combination, they result in a better control of the engineering process leading to the development of systems and products. OpenComRTOS plays an important role in this approach as it is the system software layer allowing the mapping of the abstract interacting entities at the modeling level into concrete concurrent instances.

1.1 Introduction

Our economy, our social and political environment can be considered as a system of systems. As citizens, we want these systems to work for us and to improve our lives. Technology and engineering are playing a growing and important role in it. The main reason for this fact is that technology allows us to do more with less. Technology provides us with efficiency. The task of the engineer is to put technology at work and to develop systems and products that provide us with added value. This applies to many domains, even in domains where technology only plays a supporting role and the role of the human is still dominant.

The authors of this book are mostly concerned with the domain of so-called embedded systems. While there is no unique definition for this domain, think about it as the domain of devices and systems that have a processor and software inside, often fully invisible to the user. It came into being when the transistor was invented. This was the start of the digital electronics era. Digital implies that it became more and more practical for engineers to start building systems based on the concept of state machines. What the solid-state transistor changed was that because of its shrinking size, many of these components could be used together to build very

E. Verhulst et al., *Formal Development of a Network-Centric RTOS: Software Engineering for Reliable Embedded Systems*, DOI 10.1007/978-1-4419-9736-4_1,

large scale state machines. A typical example is the processor in a desktop PC, now containing several of such devices, each having close to a billion transistors. Even a small processor can contain a few 10,000 to a few 100,000 transistors. On top of that, engineers made these components programmable. This comes down to using components whose functionality changes all the time (essentially at the rate of their clocks, often measured in MHz or GHz). The programs they run are composed of elementary instructions, meaning that the use of programs increases the size of the state machine exponentially. How do we ensure that such systems can be trusted to be correct?

This is not an easy task. Before electronics, most systems were analogue or mechanical ones. Such systems often require a lot of energy and are bulky, but usually they are quite trustworthy. The reason for this is that such systems inherently provide what is called "graceful degradation". Their state space is continuous and hence infinite, but when the material properties are affected by e.g. wear and tear, a mechanical system will keep delivering its function, even when it will have become less efficient. This is the property of graceful degradation. Of course, at some state, the system will break down as well, but there will be ample warning (if one cares to look and listen).

Digital electronic systems are often designed and manufactured in such a way that each individual transistor remains in a safe domain over its anticipated lifetime, just like with mechanical system. The difficulty comes from the fact that in an electronic system, these transistors are connected and therefore they create a large state machine. When a single transistor or its connections to another transistor fails for some reason, the system might continue to work but there is also a non-zero probability that the failure will bring the whole system to a halt. Often this means it goes into an illegal, read: undefined, state. Fortunately, in (small) digital electronics the state space is still combinatorial and in principle, one can simulate the system across all these states or one can even design a test set-up that will exercise all possible states, allowing to verify that the design prevents the system from reaching such an illegal state, even if such an event is very unlikely under normal operating conditions. The issue is that reaching such an illegal state can become very likely when the operating conditions are no longer "normal" (e.g. because the external conditions put the device outside its normal operating conditions). Often, the result will be catastrophic.

The problem really becomes horrendous when we look at embedded software running on such an electronic component. The issue is that now the size of the state space is exponentially expanded. This is partly due to the way software instructions are encoded in the hardware. If a single bit is changed, the behaviour can become entirely different. In addition, programmable electronic components are often built as so-called von Neumann machines. The processor instructions are executed in sequence. The program will also contain branching points, meaning that the resulting state space can grow very large, even under normal operating conditions. Moreover, embedded software will often not have the property of graceful degradation. If for some reason the next instruction is not the right one, the system can come to a halt in nanoseconds and standard processors cannot recover

from such errors. A hard reset and rebooting from the beginning is often the only sensible option. Most of us are familiar with this notion, often called a "blue screen", but very few know that an ordinary PC will have at least one memory bit flipped per day due to cosmic radiation. While this is often innocent, when such an event occurs in a safety critical system, lives can be at stake.

Given that the state space is now exponential and that it is physically impossible to test all possible states, how can we then have confidence in embedded software? The solution engineers adopt is to prove that the software will be correct (this holds under the assumption that the hardware is correct as well). This is essentially not different from what engineers do in other domains. For example, construction and material engineers will often not test their construction to see when it will fail. No, they will develop a mathematical model and calculate the breaking point based on the assumption that their raw materials were correctly manufactured. This allows them to apply a hefty safety margin to their design. Unfortunately, software cannot be made robust by adding somewhere a safety margin, hence we must "calculate" it exactly. This is what the emerging field of formal techniques is all about and this book is about its application to the development of a crucial embedded software component: a network centric Real-Time Operating System.

Another aspect is that the development of embedded software is not a "standalone" activity. Embedded application software has many dependencies, often on third party input or components. In addition, embedded software is essentially implementing a real-world context as a computer program. If the description of this real-world context is erroneous, these errors will be found back in the resulting application software and there they can result in erroneous products even if the implementation of the software was done correctly.

Therefore, we need to look at the whole systems engineering process. This is essential to develop trustworthy products because engineering a product involves a lot of human activity. It is a complex process with many aspects and many problems that need to be mastered. One of them is the use of natural language. Because natural language is not precise enough, often vastly differing between cultures and different domains, it is the source of many issues in systems engineering. Therefore, we must try to achieve a common language across all domains that are involved in the engineering of a product or a system. We called this trying to achieve "unified semantics". The only way to do this is to develop a unified "systems grammar" as we call it, that covers the full domain of systems (or software) engineering. This is similar to the development of an ontology but it adds the notion of "interaction" to make the relationships between the concepts concrete from early requirements to the final release of the product or system being developed.

Just like in a language it defines terms of a vocabulary and relationships between these terms. Such a systems grammar will also seek orthogonality, essentially trying to come up with terms that have no overlapping and no ambiguous meaning. Less is often better in this context. It can be understood as an application of Einstein's principle (or occam's razor if you prefer). Keep things simple, but not too simple. Essentially, if a solution is complex, it is not because its creators were smart, but because they did not fully understand the problem at hand. Below follows a