

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. Spath
Vitaliy Mezhuyev

Formal Development of a Network-Centric RTOS

Software Engineering for Reliable Embedded
Systems

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. Spath
Open License Society
Zavelstraat 160
3010 Leuven
Belgium
bernhard.spath@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

© Springer Science+Business Media, LLC 2011

All rights reserved. This work may not be translated or copied in whole or in part without the written permission of the publisher (Springer Science+Business Media, LLC, 233 Spring Street, New York, NY 10013, USA), except for brief excerpts in connection with reviews or scholarly analysis. Use in connection with any form of information storage and retrieval, electronic adaptation, computer software, or by similar or dissimilar methodology now known or hereafter developed is forbidden.

The use in this publication of trade names, trademarks, service marks, and similar terms, even if they are not identified as such, is not to be taken as an expression of opinion as to whether or not they are subject to proprietary rights.

Printed on acid-free paper

Springer is part of Springer Science+Business Media (www.springer.com)

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 Spath 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	xv
List of Tables	xvii
Part I Trustworthy Embedded Systems	
1 Introduction: OpenComRTOS Role in a Unified Systems Engineering Methodology	3
1.1 Introduction	3
1.2 A Systematic Engineering Methodology Based on Unified Semantics and Interacting Entities.....	6
1.3 Interacting Entities for the Software Domain	9
1.3.1 Silicon Technology Advances	10
1.3.2 Silicon Technology Limitations	10
1.3.3 The World Becomes Connected	11
1.4 A Link with the Work Plan in a Systems Engineering Project	11
1.5 System Engineering Methods and Engineering Standards.....	12
1.6 Where Do Formal Techniques Fit in?.....	13
2 Requirements and Specifications for the OpenComRTOS Project	15
2.1 Background of OpenComRTOS	15
2.2 Early Requirements Derived from the Virtuoso RTOS	17
2.3 Real-Time Embedded Programming	19
2.3.1 Why Real-Time?	19
2.3.2 Why a Simple Loop Is Often not Enough.....	20
2.3.3 Superloops and Static Scheduling	21
2.3.4 Rate Monotonic Analysis	24
2.3.5 Priority based Scheduling in OpenComRTOS	26
2.3.6 The Issue of Priority Inversion and Its Inadequate Solution	27
2.4 Next Generation Requirements	29

2.5	Top Level Requirements for OpenComRTOS	32
2.6	Specifications Derived from Requirements	34
2.7	Systems and Application Grammar of OpenComRTOS	36
2.7.1	Base Principles and Definitions	36
2.7.2	A Note on Typing Conventions	37
2.7.3	OpenComRTOS System and Application Grammar	37
 Part II Formal Modeling Fundamentals		
3	The Choice of TLA⁺/TLC: Comparing Formal Methods	45
3.1	Formal Methods Survey and Pre-Selection	45
3.2	Case Study	46
3.2.1	Introduction	47
3.2.2	The Algorithm	47
3.2.3	Remarks	49
3.2.4	Drawbacks	49
3.2.5	Related Work	50
3.3	TLA ⁺ and TLC	51
3.3.1	Overview	51
3.3.2	Model Developed	53
3.4	Promela and SPIN	59
3.4.1	Overview	59
3.4.2	Model Developed	62
3.5	Comparison	66
3.5.1	Matching of the Method to the Application	66
3.5.2	Human Factors	66
3.5.3	Widespread Utilization	66
3.5.4	Licensing/Distribution	68
3.5.5	Maturity	68
3.5.6	Performance	68
3.5.7	Interface	68
3.5.8	Coverage of the Input Language	68
3.5.9	Bibliography	69
3.5.10	Expressiveness	69
3.5.11	Readability	70
3.5.12	Reusability	71
3.5.13	Scalability	71
3.5.14	Level of Abstraction	71
3.5.15	Checking Possibilities	72
3.5.16	Coverage of the Lifecycle	72
4	Basic Formal Specification in TLA⁺	73
4.1	Introduction	73
4.1.1	Goal: Awareness in Specifying Systems	73
4.1.2	A Two-Step Approach	73

- 4.2 Structure of TLA⁺ Specifications 74
 - 4.2.1 Basic Structure 74
 - 4.2.2 Module Structure 75
- 4.3 Introducing TLA⁺ By Example 76
 - 4.3.1 Basic TLA⁺ Notions 76
 - 4.3.2 Basic Examples: TLA⁺ Sequences
and OpenComRTOS Lists 77
 - 4.3.3 An Extended Example: The Module *Port* 79
- 4.4 Conclusion 85

Part III OpenComRTOS Design

- 5 Formal Modelling of the RTOS Entities** 89
 - 5.1 Introduction 89
 - 5.2 OpenComRTOS Environment Model 90
 - 5.2.1 Term Definitions 91
 - 5.2.2 Constants 91
 - 5.2.3 Variables Representing the System State 92
 - 5.2.4 The L1-Packet 92
 - 5.2.5 General Constraint for All Models 93
 - 5.3 Formal Model of the Semaphore-Entity 93
 - 5.3.1 Constants 94
 - 5.3.2 Variables 94
 - 5.3.3 Initialisation 94
 - 5.3.4 Signalling the Semaphore 95
 - 5.3.5 Testing the Semaphore 97
 - 5.3.6 Constraints 100
 - 5.3.7 Defining the Next State 101
 - 5.3.8 Properties to Check 101
 - 5.3.9 Proof Obligations 102
 - 5.3.10 Checking the Models 104
 - 5.4 Model Verification 104
 - 5.5 Conclusion 105
- 6 Final Architecture of the RTOS** 107
 - 6.1 The Building Blocks of OpenComRTOS 107
 - 6.1.1 The Hub Entity of OpenComRTOS 108
 - 6.1.2 Tasks 113
 - 6.1.3 Packets 114
 - 6.2 The Semaphore Loop 115
 - 6.2.1 The Semaphore Loop in Detail 116
 - 6.2.2 Heterogeneous Multiprocessor Systems and
Their Issues 118
 - 6.3 OpenComRTOS Development Process for Applications 119
 - 6.4 Summary 119

7	Task Interaction Models in OpenComRTOS	121
7.1	Introduction	121
7.2	Modelling Task Interaction	123
7.3	Timing Properties of Task Interactions	126
7.4	Notes on Asynchronous Interactions	128
7.5	Conclusions	131
8	Results: Code Size and Performance	133
8.1	Metrics of Success	133
8.1.1	Code Size	134
8.1.2	Total Memory Use	137
8.1.3	Influence of Processor Architecture	137
8.1.4	Semaphore Loop	139
8.1.5	Interrupt Latency	139
 Part IV Appendix		
A	OpenComRTOS-Suite 1.3 Usage Tutorial	143
A.1	Developing a Single Node Semaphore-Loop Project	143
A.2	Going Distributed with OpenComRTOS	153
A.3	Tracing in OpenComRTOS	155
A.3.1	How to Enable Tracing	157
A.3.2	How to Retrieve a Trace	158
A.3.3	Retrieving and Displaying Traces from Distributed Systems	159
A.4	Measuring the Interrupt Latency of OpenComRTOS	160
A.4.1	Designing Distributed Heterogeneous Systems Using the OpenComRTOS Suite	160
A.4.2	Presenting the Measurement Results	162
A.4.3	Specifying the System	162
A.4.4	Implementation	164
A.4.5	Application	165
A.4.6	Collected Measurement Results	166
A.5	Summary	168
B	Foundations for TLA⁺ and Temporal Logic	169
B.1	Introduction	169
B.1.1	Goal: Increased Awareness in Specifying Systems	169
B.1.2	Approach and Overview	170
B.2	A Unifying Formalism	171
B.2.1	Rationale	171
B.2.2	Syntax	171
B.2.3	Style of Use	173
B.2.4	Introducing TLA ⁺ Via Funmath	178
B.3	Faithful Formalization of Informal Specifications	179
B.3.1	Choice of Proper Data Abstractions	181

- B.3.2 Auxiliary Functions in Formal Specifications 184
- B.4 Calculational Reasoning and Patterns in TLA^+ 186
 - B.4.1 Capturing Temporal Logics by Temporal Calculi 186
 - B.4.2 A Functional Temporal Calculus (FTC) 187
 - B.4.3 Defining the *Temporal Calculus of Actions* (TCA) 190
 - B.4.4 Calculational Reasoning in TCA/ TLA^+ 192
 - B.4.5 Applications to Patterns in TLA^+ 194
- B.5 Conclusions 196
- C Comparison of Formal Methods** 199
 - C.1 TLA^+ Model of Harris' Algorithm 199
 - C.2 Promela Model of Harris' Algorithm 206
- Glossary** 211
- References** 213
- Index** 217

List of Figures

Fig. 1.1	Unified Systems Engineering Methodology	7
Fig. 2.1	The context of systems engineering	16
Fig. 2.2	Superloop schedule with three interrupt sources	22
Fig. 2.3	Two periodic tasks scheduled with RMA	24
Fig. 2.4	Three tasks sharing a resource with and without priority inheritance support	30
Fig. 3.1	Example of a specification in TLA ⁺	52
Fig. 3.2	Header of the TLA ⁺ model	56
Fig. 3.3	Bottom of the TLA ⁺ Specification of Harris' algorithm	57
Fig. 3.4	Definition of <i>Coherence</i>	57
Fig. 3.5	Definition of action <i>CreateI</i>	57
Fig. 3.6	Definition of action <i>LocateD</i>	58
Fig. 3.7	Definition of action <i>CasD2</i>	59
Fig. 3.8	Comparison of the formalisms	67
Fig. 4.1	Rendezvous in <code>SendReceivePacketService</code>	83
Fig. 6.1	A simple OpenComRTOS application using Port Hubs.....	108
Fig. 6.2	Hub diagram.....	109
Fig. 6.3	Application diagram with all Interactions for the Semaphore Loop	115
Fig. 7.1	Task interactions with a Hub	124
Fig. 7.2	Asynchronous Task interactions with Hubs	129
Fig. A.1	Screenshot of OpenVE's the 'New Project' dialogue	144
Fig. A.2	OpenVE with opened Topology View (no nodes defined yet)	145
Fig. A.3	The dialogue to specify the properties of the new win32-node	145

Fig. A.4 Topology view showing the newly created win32-node 146

Fig. A.5 OpenVE Application Diagram with highlighted Task button 146

Fig. A.6 ‘New Task’ dialogue, with highlighted ‘Task
Entrypoint’ creation button 147

Fig. A.7 The Task Entrypoint creation dialogue, showing the
source code that will be generated 147

Fig. A.8 Application diagram showing the newly created Task1 148

Fig. A.9 Application diagram showing both Task1 and Task2 149

Fig. A.10 The ‘New Semaphore’ dialogue of OpenVE 149

Fig. A.11 Application diagram with all entities, showing the
interaction selection menu 150

Fig. A.12 Application diagram with all Interactions for the
Semaphore Loop 150

Fig. A.13 Source code for Task1, the incorrectly placed
interactions highlighted 151

Fig. A.14 The ‘New stdioHostServer’ dialogue of OpenVE 151

Fig. A.15 Application diagram with the complete Semaphore-
Loop and the Stdio Host Server 152

Fig. A.16 Source code of Task1 with Semaphore and Stdio Host
Server Interactions 152

Fig. A.17 Console output upon running the ‘SemaphoreLoop’ project 153

Fig. A.18 Edit Link Ports Dialogue, with highlighted ‘Add Link
Port’ button 154

Fig. A.19 OpenVE link configuration dialogue 154

Fig. A.20 Topology of the two Win32 Nodes connected with a
bidirectional Link 155

Fig. A.21 OpenVE with open ‘Properties’ side-pane and
highlighted ‘node’ property 156

Fig. A.22 Console output of both win32-nodes 156

Fig. A.23 OpenVE with open Property Pane 157

Fig. A.24 Tracing enabled Application Diagram 159

Fig. A.25 OpenTracer displaying parts of the
SemaphoreTracing_MP_TCPIP example trace 159

Fig. A.26 Stages of IRQ handling in a typical Microcontroller System 161

Fig. A.27 Screenshot of the Interrupt Latency GUI Application 164

Fig. A.28 Interrupt latency measurement system topology 165

Fig. A.29 Interrupt latency measurement system application diagram 166

Fig. A.30 Measured IRQ to ISR Latency on ARM Cortex M3
50MHz (logarithmic scale) 167

Fig. A.31 Measured IRQ to Task Latency on ARM Cortex M3
50MHz (logarithmic scale) 167

Fig. B.1 Numbers of beans 182

Fig. B.2 Bag of beans 183

Fig. B.3 The *Bags* module from *Specifying Systems* compressed 185

List of Tables

Table 3.1	Correspondence of TLA ⁺ model with the textual description in Sect. 3.2.2	57
Table 7.1	Time semantics of two Tasks interacting in OpenComRTOS	126
Table 8.1	Code size for OpenComRTOS kernel on MLX16	135
Table 8.2	Total memory used for OpenComRTOS on MLX16	138
Table 8.3	Code size figures (in 8 bit Bytes)	139
Table 8.4	Semaphore Loop times (=2 signals, 2 tests, 4 context switches) in microseconds	139
Table B.1	Basic mathematical TLA ⁺ expressions via Funmath equivalent, part 1	179
Table B.2	Basic mathematical TLA ⁺ expressions via Funmath equivalent, part 2	180
Table B.3	Action and temporal operators of TLA ⁺ defined via Funmath ...	180

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

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