UNIVERSIDADE DA BEIRA INTERIOR
Covilhã | Portugal

Towards a Formally Verified Microkernel using the VCC Verifier

Joaquim José e Silva de Carvalho Tojal

Submitted to University of Beira Interior in candidature for the degree of
Master in Computer Science

Supervised by Simão Melo de Sousa
Co-supervised by José Miguel Faria

Department of Computer Science
University of Beira Interior
Covilhã, Portugal
http://www.di.ubi.pt

Acknowledgements

I would like to express my deep thank to all those who in one way or another, contributed to the achievement of this work, in particular:

To my supervisor, Professor Simão Melo de Sousa, for scientific guidance and support that has always had, without which the achievement of this work would not be possible.

To my co-supervisor, José Miguel Faria, for the availability, interest, guidance and friendship that has always had and that was very important to the success of this work.

To André Passos by the friendship and constant support.

To all my colleagues in Critical Software who helped me with xLuna.

To Critical Software for giving me the opportunity to develop this thesis in a company like that, which was a very enriching experience.

To my girlfriend, Rita, for the patience and care she had for me throughout this period, for the help, support and constant encouragement.

To my parents, Marilia and Adriano, and my brother, Ricardo, for their trust and care that always gave me and for all the support, because without it would never have gotten where I am.

For all of you: Thanks!

Abstract

In this thesis we present the design by contract modular approach to formal verification of an industrial real-time microkernel which was not designed with formal verification in mind. The microkernel module targeted is a particular interrupt manager of xLuna Real Time Operating System (RTOS) for embedded systems built by Critical Software S.A. The annotations were verified automatically using the Microsoft Research Verified C Compiler (VCC) tool to reason about concurrency and safety properties of xLuna kernel. The specifications are based in Hoare-style pre- and post-conditions inlined with the real code.

xLuna is a microkernel based on the RTEMS Real-Time Operating System. xLuna extends RTEMS for run a GNU/Linux Operating System, providing a runtime multitasking environment for real-time (RTEMS) and non-real-time (Linux) applications.

xLuna runs in a preemptable and concurrent environment. Therefore, we use VCC for reasoning about concurrent executions and some functional and safety properties of xLuna microkernel. VCC is an automated verifier for concurrent C programs that is being developed by Microsoft Research, Redmond, USA and European Microsoft Innovation Center (EMIC), Aachen, Germany. VCC is being built and used for operating system verification which makes it suitable for our verification work.

Specifications were added to xLuna code following a modular approach to the verification of a specific microkernel module, namely the Interrupt Request (IRQ) module. The Verified C Compiler (VCC) annotations added cover approximately 80% of the IRQ manager C code (the remaining 20% of the code are relative to auxiliary functions outside the scope of our verification work). All the annotations were automatically verified and proven to be correct.

Keywords

Concurrency, Critical Systems, Design By Contract, Embedded Systems, Formal Verification, Formal Methods, Microkernel, Real-Time, Operating System, Safety, Security, Software Reliability, Verified C Compiler, xLuna.

Contents

List of Figures

Acronyms

API Application Programming Interface

CC Common Criteria

CPU Central Processing Unit

CVM Communicating Virtual Machines

DFKI Germen Research Center for Artificial Intelligence

DSU Data Secure Unix

EAL Evaluation Assurance Level

EMIC European Microsoft Innovation Center

ESA European Space Agency

HDM Hierarchical Development Methodology

HRT Hard Real Time

IDE Integrated Development Environment

IEC International Electrotechnical Commissions

IPC Inter-Process Communication

IRQ Interrupt Request

ISC Inter-Systems Communication

JML Java Modeling Language

KIT Kernel for Isolated Tasks

KSOS Ford Aerospace Kernelized Secure Operating System

LOC Lines of Code

MASK Mathematically Analyzed Separation Kernel

MILS Multiple Independent Levels of Security

MMU Memory Management Unit

MSc Master of Science

NASA National Aeronautics and Space Administration

NICTA Australia's ICT Research Center of Excellence

NRT Non-Real-Time

NSA National Security Agency

OS Operating System

PSOS Provably Secure Operating System

PVS Prototype Verification System

RAMS Reliability, Availability, Maintainability and Safety

RTEMS Real-Time Executive for Multiprocessor Systems

RTOS Real Time Operating System

seL4 Secure Embedded L4

SKPP Separation Kernel Protection Profile

SPECIAL Specification and Assertion Language

SIPs Software-Isolated Processes

SRI Stanford Research Institute

SRMMU SPARC Reference Memory Management Unit

SOS	Simple Operating System
TAME	Timed Automata Modeling Environment
TLS	Top Level Specification
UCLA	University of California at Los Angeles
USA	United States of America
VCC	Verified C Compiler
VFiasco	Verified Fiasco
xLuna	eXtending free/open-source reaL-time execUtive for oN-board space Applications

Chapter 1

Introduction

The subject of this thesis is the formal verification of operating systems using automated and mathematical-based tools to prove software correctness. This work is integrated in the context of the Master of Science (MSc) course in computation an intelligent systems. This chapter introduces the main aspects related to this work.

1.1 Motivation

Almost every computer system depends directly on the operating system behavior. Having kernel code that is proved to be correct is a goal that researchers and industrial company's have attempted to achieve. Large amounts of low-level implementations like operating system core are obviously a perfect and challenging target for formal verification.

Nowadays, formal software verification or development is seen as the best way to achieve a greater quality in software systems. Mainly in critical systems, software reliability for safety and security is a crucial requirement. For instance DO-178B [1] standard for avionics industry, demands extensive validation, careful test cases design, requirement engineering and coverage analysis. Indeed, The International Electrotechnical Commissions (IEC) standard IEC 61508 (SIL 4) [2] and DO-178B (Design Assurance Level A and B), recommend the use of formal methods. However, an even more demanding standards for security evaluation is the Common Criteria (CC) Evaluation Assurance Level (EAL) framework [3]. CC EAL security evaluation provides seven assurance levels (EAL-1 to EAL-7). The last three levels require the use of formal methods in the development process.

- EAL 1,2,3 and 4. For this 4 levels its not required the use of formal methods in software requirements, functional specification, system high-level design, low-level design and implementation. CC EAL-4/5 have been compared with DO-178B.
- For EAL-5 formal methods shall be applied in requirements phase and semi-formal methods for functional specification and system high-level design.
- EAL-6 is EAL-5 plus semi-formal verification for low-level design.
- Finally the last, and more demanding standard, EAL-7 requires the use of formal methods in software requirements, functional specification, system high-level design and also semi-formal verification for low-level design.

One can see that none of these levels requires a formal implementation. A fully verified system is one that uses formal methods at all levels. Therefore modular verification of each function at the code level could be considered an efficient way prove implementation correctness since for CC assurance framework, functions are interpreted as black boxes. In our approach, annotations are inlined within function implementation to prove that the specification meets the implementation.

The specifications follow the $design - by - contract$ [4] paradigm based in Hoare logic [5] with pre- and post-conditions. Basically, each function has pre-condition and a post-condition that can be viewed as a contract between the same function and its caller. The source code is then confronted with that contract. This type of formalism is most of the times closely connected to the implementation language used. For C code we have Frama-C [6] and VCC [7] (the tool used in this thesis), for Java we have Java Modeling Language (JML) [8], for C# we have Spec# [9] and for Ada we have Ada/Spark [10] are some examples of tools that provide support for software verification.

1.2 Context

This work is inserted in a partnership between Critical Software,SA.[11] and the University of Beira Interior. The knowledge transfer process between these two entities was important during the development of this work. Our verification target is the Critical's xLuna Real-Time microkernel for embedded systems. Particularly, a special xLuna interrupt request manager was chosen for verification.

As verification tool we chose the Microsoft's VCC. VCC has been built mainly for operating systems verification with particular support for reasoning about concurrent environments which fits perfectly on the verification target. VCC was already been used for this purpose in real industrial system as xLuna. Namely, in Microsoft Hypervisor and PikeOS microkernel.

1.3 Work Definition and Objectives

This Work was divided in four main steps. (i) First it was necessary to be aware of all the related work that had already been done or still in development. After that (ii) a careful study of the verification tool, verification methodology and xLuna microkernel was made. Then, we had to (iii) connect the verification tool environment to the source code by performing appropriate code isolations to maintain a manageable scope. Finally (iv) the specifications were added to xLuna code following a modular approach to the verification of a specific microkernel module (IRQ).

Now we list the main objectives of this work:

- Use and explore the VCC tool to perform an appropriate application.
- Complete verification of xLuna IRQ module C code.
- Achieve a basis for a possible certification.

1.4 Contribution

Formal verification of xLuna IRQ module was performed on two parallel paths using two different tools. This work is one of these paths. The results of the two approaches have been submitted to publication with an article [12] showing, in a comparative sense, the two experiences.

Beyond publication for the related scientific community and the know-how acquired within Critical Software, verify a realistic and industrial system like xLuna brings a major value to the product and for the company taking into account a future certification.

1.5 Outline

The remaining of this document is organized as follows:

- Chapter 2 presents an overview on VCC tools and verification methodology.
- Chapter 3 describes the xLuna architecture and some particular design aspects.
- Chapter 4 describes the related work in operating system verification.
- Chapter 5 exposes the detail design of xLuna IRQ manager and the verification work for each function.
- Chapter 6 presents the verification results and future improvements.

Chapter 2

VCC: Verified C Compiler

VCC is a fully automated verification tool for concurrent C that is being developed by Microsoft Research, Redmond, United States of America (USA) and EMIC, Aachen, Germany [7]. VCC utilizes annotations based in Hoare-style pre- and post-conditions closely attached to source code. That is, the annotations are mixed into the *codebase* rather than within blocks of commented (for the C compiler) specifications. Even so, the annotated C code can still be compiled with any C compiler through conditional compilation: If a regular C compiler is called, a special VCC flag is disabled and the annotations and specification code are preprocessed and transformed into empty strings. The concept of complete code with annotations is the same used in JML [8] for Java code or SPEC# [9] for C# programs.

VCC performs a static modular analysis, which scales for verification of bigger systems like an operating systems. One function can thus be verified in isolation without knowing its use or possible callers [7].

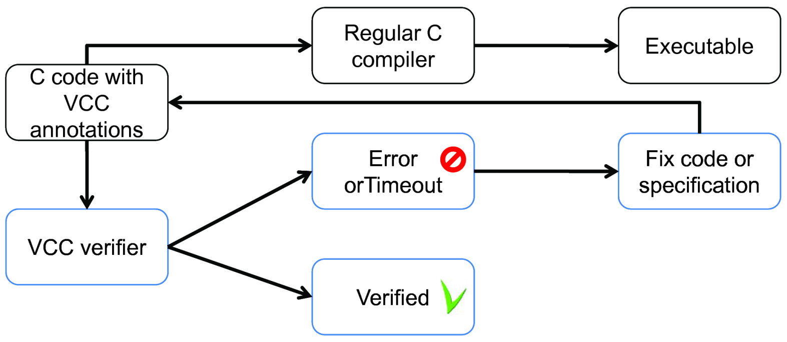


Figure 2.1: VCC Workflow

2.1 Verification Toolchain

VCC is fully integrated with the well known Microsoft Visual Studio Integrated Development Environment (IDE), providing a familiar environment to programmers and making the correct use of the VCC tool-chain very simple. One of the advantages of having an integration with familiar tools allow developers to verify programs while implementing. Three main steps are made by VCC when trying to verify an annotated C program:

- (i) VCC translates annotated C code into BoogiePL (an intermediated verification language), asserting for instance, type-safe memory access, arithmetic overflows, etc.

- (ii) Boogie translates BoogiePL into first-order predicate formulas, which are a number of mathematical statements more often called as verification conditions.

- (iii) Z3 tries to solve them and if it finds a proof then the program is correct according to the specifications. If Z3 can not find a proof it generates a counter-example with a sequence of program states and variable assignments that can be viewed using the VCC model viewer. However, memory or time resources are limited and the prover may fail without a counter-example. For this case VCC provides the Z3 visualizer which show what the prover was trying to do when run out of resources [13]. Figure 2.1 shows the verification flow for using VCC.

2.2 VCC Annotations

As in standard *design by contract* approach to partial correctness, the pre- and post-conditions inserted into a C function constitute a contract between the function and a function caller, guaranteeing that if the function starts in a state that satisfies the precondition then the postcondition holds at the end of the function.

To express and reason about specifications, VCC uses clauses such *requires* to specify in which condition the function can be called, *ensures* clause to specify in which condition the function is allowed to return, *result* function to reasoning about the function return value or *writes* representing the frame conditions which limit what the function is authorized to modify. VCC also supports loop invariants for reasoning about loop behavior.

2.3 Ownership, Type Invariants

VCC memory model [14] ensures that objects (pointers to structures) do not overlap in memory, keeping a typed and hierarchical view of all objects (Spec# ownership model).

For instance, one may need to specify all the locations written by a function (e.g. fields of an object). This breaks data encapsulation and thus, VCC implicitly lets the object own its representation and writing the object allows writing its whole ownership domain. However, updating an object requires a special $wrap$/$unwrap$ protocol to transfer ownership domains [15]. Figure 2.2 object transitions during this protocol.

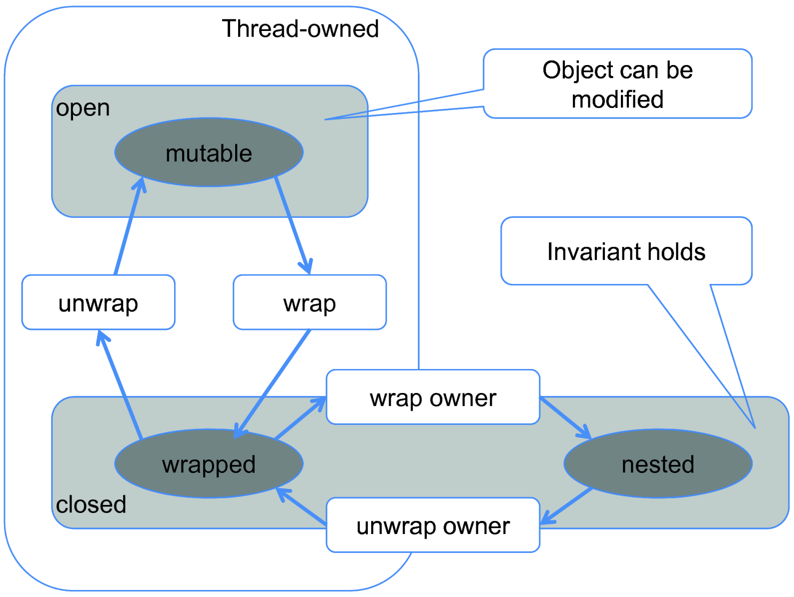


Figure 2.2: VCC wrap/unwrap protocol

Stating an hierarchical structure (in specification code) gives VCC a tree view of the system. One object can only have one owner and threads own themselves. One can see in Figure 2.2 that a specific ownership domain ideally it should be $open$ or $close$ inside of the thread domain and close outside the thread domain. In a sequential fashion an object can only be updated when it is $mutable$ and must return to an safe state performing some operations required by the $wrap$/$unwrap$ protocol. The explanation of these states is given bellow.

- Mutable
  - After creation the object is open and owned by $me()$ which represents the current thread.

- At this state the thread is allowed to modify the object and prevents other threads interference.

- Wrapped
  - Wrapping the object ($wrap$) requires its invariant to hold.
  - The reverse transition ($unwrap$) assume object invariants.

- Nested
  - Closed objects can be added to (or removed from) another objects ownership via $set\_owns()$, $set\_closed\_owner()$ (wrap owner) or $giveup\_closed\_owner()$ (unwrap owner) operations.
  - The reverse transition ($unwrap$) assume object invariants.

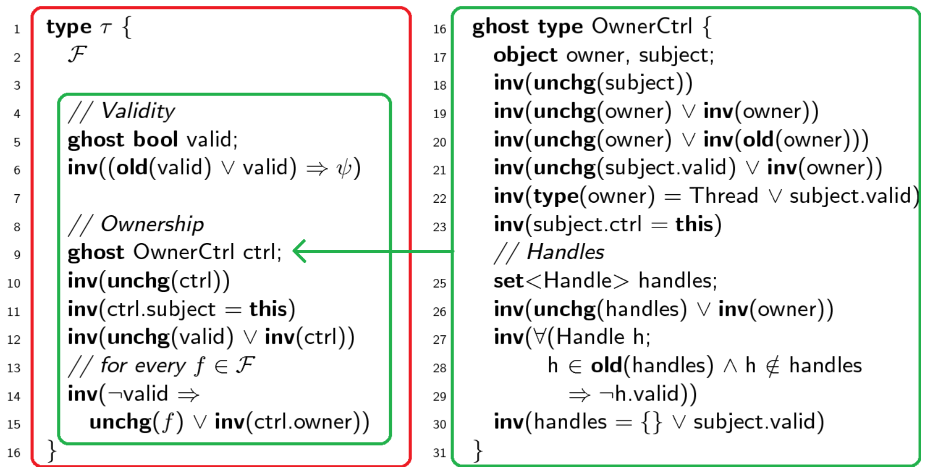
Structures can be also annotated with type invariants related to its ownership behavior or to its own fields. This can be useful to state safe object transitions in concurrent context. Moreover, VCC implicitly applies an object transformation to each object adding implicit invariants that are checked during this protocol.

2.4 Ghost Code

This ownership machinery is applied as a *ghost* transformation behind every structure in the program. Each type has a related ownership control object in specification code (*ghost* code), only visible to VCC and providing a bridge between implicit specification code and VCC annotations in the real program. This is a crucial concept in verification methodology. Figure 2.3 shows the transformation applied to each structure.

This support for communication between programmer and verifier is an hidden (Ghost state) verification mechanism and cannot modify non-ghost state. In transformation seen in Figure 2.3 type T have its own fields F. VCC implicit adds ghost fields, ownership control objects (not transformed) and invariants (see [16]). This ghost code is the code inside the green rectangles in Figure 2.3.

Ghost code can also exist as explicit specification functions, objects or variables only seen by VCC for verification purposes (surrounded with $spec()$ clause).



```
type τ {
  𝓕

  // Validity
  ghost bool valid;
  inv((old(valid) ∨ valid) ⇒ ψ)

  // Ownership
  ghost OwnerCtrl ctrl;
  inv(unchg(ctrl))
  inv(ctrl.subject = this)
  inv(unchg(valid) ∨ inv(ctrl))
  // for every f ∈ 𝓕
  inv(¬valid ⇒
    unchg(f) ∨ inv(ctrl.owner))
}

ghost type OwnerCtrl {
  object owner, subject;
  inv(unchg(subject))
  inv(unchg(owner) ∨ inv(owner))
  inv(unchg(owner) ∨ inv(old(owner)))
  inv(unchg(subject.valid) ∨ inv(owner))
  inv(type(owner) = Thread ∨ subject.valid)
  inv(subject.ctrl = this)
  // Handles
  set<Handle> handles;
  inv(unchg(handles) ∨ inv(owner))
  inv(∀(Handle h;
        h ∈ old(handles) ∧ h ∉ handles
          ⇒ ¬h.valid))
  inv(handles = {} ∨ subject.valid)
}
```

Figure 2.3: VCC type transformation

2.5 VCC clauses

It is important to understand other key annotation function given by VCC and also used in this work. A list of those annotations are given bellow.

- $keeps()$. Used mostly in ownership invariants to state fixed ownership sets. For instance, an structure with the invariant $invariant(keeps(\&objectA, \&objectB)$ will always own just this two objects. It is also used when dealing with array objects.
- $spec()$ or $speconly()$. For specification variables and specification code respectively (invisible to the compiler).
- $set\_owns(A, B)$. Updates the owner set of A adding B. This statement will only be asserted when wrapping A.
- $set\_owner(B, A)$. Union of the old owns set of A and the object B.
- $thread\_local(A)$. A points to an object that belongs to the current thread.
- $maintains()$. pre and postcondition with the same predicate (as a function invariant).
- $always(A, closed(B))$. Maintains A $wrapped$ and A has a implicit invariant which implies $closed(B)$.
- $atomic(objs)$. VCC requires updates to volatile fields of closed objects to be within atomic block. VCC checks the invariants of each object in $objs$ over this action,

and that are all closed. Defines the safe transition of the objects when updating their volatile fields.

2.6 Summary

This chapter presented an overview of the VCC verification methodology and tools. VCC has been built to target low-level C code in a concurrent environment. Therefore, suitable for our verification target. The next chapter shows the xLuna real-time operating system architecture.

Chapter 3

xLuna Real-Time Operating System

Most of the space missions software is based on a RTOS solution. Although, some recent space applications require a more high level support and standard development environment such as Linux operating system. Recent space applications with new autonomy requirements, mapping or navigation demand less low-level services like file system or a rich set of libraries that can be reused. However most of this features are not supported by some RTOS solutions. Therefore when such applications are needed, they are built in a known environment with support for high level services, tested in prototypes or flight models and then, ported to an RTOS qualified for space missions. This process requires a lot of time and undesirable effort which could be reduced using an hybrid solution.

eXtending free/open-source reaL-time execUtive for oN-board space Applications (xLuna) [17] is a microkernel based on the Real-Time Executive for Multiprocessor Systems (RTEMS) [18] Real-Time Operating System already qualified for on-board software. xLuna extends RTEMS with the ability to run a GNU/Linux Operating System [19], providing therefore a runtime environment for real-time (RTEMS) and non-real-time (Linux) applications. xLuna was designed to supports European Space Agency (ESA)'s LEON SPARC processor [17] with the main goal of extending the RTEMS kernel in order to enable a safely Linux execution without jeopardizing aspects of reliability, availability, maintainability and safety (RAMS) and at the same time providing a rich chain of compiler, debuggers, and utilities that require a lower learning curve to create applications, integrate existent components and reuse components across different missions. Its general architecture is shown in Figure 3.1.

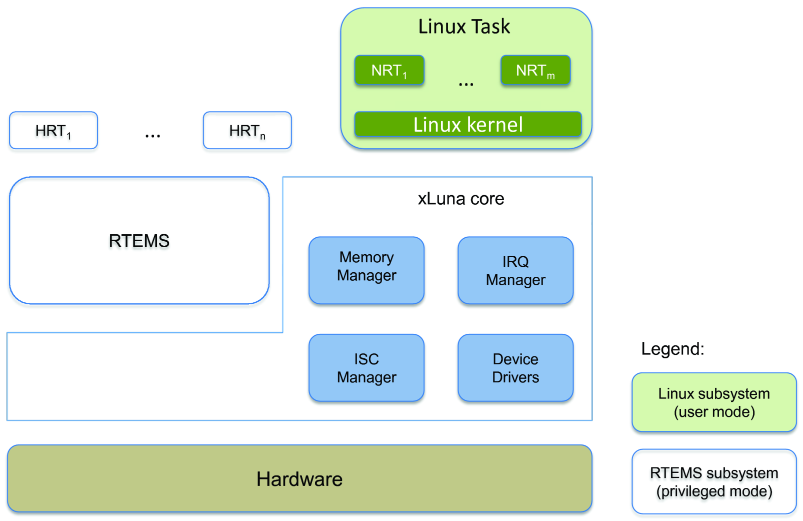


Figure 3.1: xLuna architecture

The Linux subsystem runs as an unprivileged RTEMS task and in a different memory partition. This provides spacial partitioning, guaranteeing a safe isolation between the Non-Real-Time (NRT) and Hard Real Time (HRT) subsystems. Figure 3.2 shows an example of the xLuna approach to running tasks in a secure environment. Another advantage of this architecture with two isolated subsystems is that components with less criticality level can be easily developed and added reducing time-to-market and budget. However, Linux system was designed to run with direct access to the underlying layer, and thus, violating security requirements of the whole architecture. Therefore, all Linux direct hardware accesses were eliminated and transformed in communication events caught by xLuna.

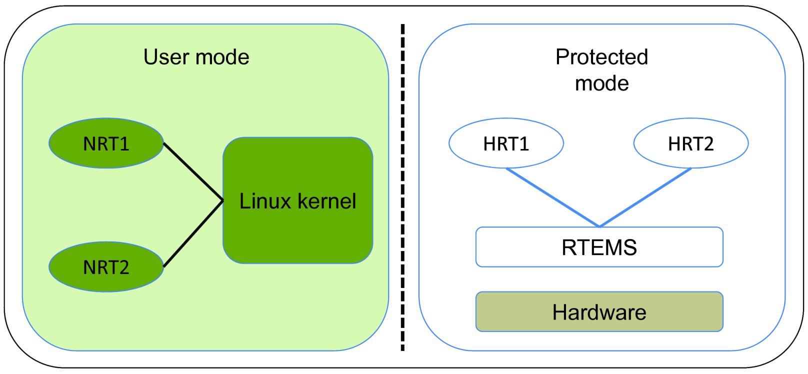


Figure 3.2: xLuna approach

The RTEMS core system was not modified, instead, were developed several xLuna managers to support the Linux integration in the overall architecture.

- Sub-systems used in xLuna project:
  - The Snapgear embedded Linux distribution, kernel version is 2.6.11.
  - RTEMS version 4.6.6.

The following sections give a brief overview of the main modules provided by xLuna microkernel.

3.1 Memory manager

Memory protection is ensured at the entry point of xLuna, during bootstrap[1] process the SPARC Reference Memory Management Unit (SRMMU) hardware is set to mapping virtual and physical address for RTEMS and Linux kernel memory windows correctly. This initial configuration enables enforcing isolation requirements between RTEMS and Linux and ensures RTEMS system protection if Linux behaves incorrectly or even if it crashes. Moreover, the memory manager xLuna module also guarantees protection between Linux kernel em Linux user processes and among Linux processes. Figure 3.3 shows xLuna system memory privileges.

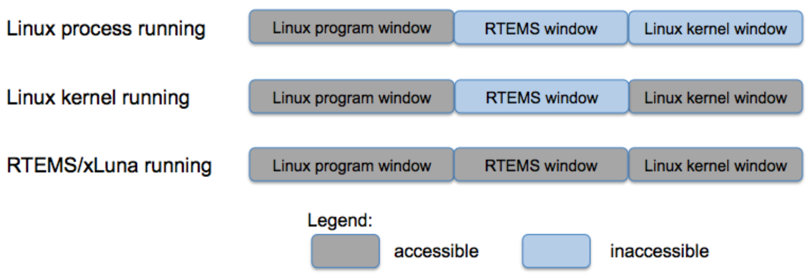


Figure 3.3: xLuna memory access rights

We can conclude from figure 3.3 that with this access rights configuration Linux processes or Linux kernel can not access to RTEMS window and Linux window is marked as invalid for Linux processes (protect Linux kernel from Linux processes).

[1]Boot sequence to load the main operating system.

3.2 Interrupt Manager

The IRQ module main goal is connect hardware interrupts/traps to the Linux kernel (which does not have access to them) and provides the services (system calls) to Linux kernel, serving as a para-virtualization[2] Application Programming Interface (API). The IRQ manager is mainly responsible for:

- Intercept hardware interrupts required by Linux subsystem and redirect them for the corresponding RTEMS handles.
- Intercept all traps (Linux system calls requested by Linux processes or xLuna system calls performed by the Linux kernel) and call the respective handlers, which can be a service from IRQ manager or other xLuna manager.
- Enable/disable interrupts virtualization for Linux kernel.

However, this methodology brings some issues when dealing with real-time and non-real-time executions. When interrupt/trap handling is working the RTEMS HRT dispatching is disabled which could affect the overall system functionality, therefore, xLuna makes sure that the time spent during this process is minimum and quantifiable for the worst case.

One can conclude form this brief description that the IRQ module is one of the most critical components of the entire xLuna system. The correct pipelining and treatment of interrupts/traps is crucial not only for Linux subsystem, but for the entire system behavior. For instance, in order to achieve communication between NRT and HRT tasks, the Inter-Systems Communication (ISC) manager uses the IRQ manager. Therefore, an IRQ malfunction may compromise both subsystems. Due to all this important factors, the xLuna IRQ manager was considered a suitable target for formal verification.

3.3 Inter-System Communication

The main goal of having an inter system communication in xLuna kernel is to provide support to Linux and RTEMS tasks communicate between them, and so, a RTEMS task (HRT) can write a message that can be read by a Linux task (NRT) and the opposite.

Such communication is performed making use RTEMS message queues located in the

[2]Software layer similar to the underlying hardware and visible to user process request.

RTEMS memory space. Since the shared resources are in the RTEMS side, communication is implemented directly using message queues.

On the Linux side, a kernel device driver is provided for inter-systems communication. It uses system calls to access the xLuna kernel services, which will call the RTEMS message queue directives. All this process is detailed in Figure 3.4 and explained in the following sections.

For the overall ISC functionality xLuna uses some resources provided by the RTEMS API. Those resources are listed bellow.

- RTEMS resources used in ISC Module:
  - 1 fixed size write memory buffer (*rtems_to_lx*)
  - 1 fixed size read memory buffer(*lx_to_rtems*)
  - 2 counting semaphores (*n_empty_linux* and *n_empty_rtems*) for write buffer control
  - 2 counting semaphores (*n_full_linux* and *n_full_rtems*) for read buffer control.

These resources are used in the communication flow of the two subsystems. The information flow and the system calls associated with the ISC manager show the importance of the IRQ module. The ISC flow and ISC IRQ usage is described bellow.

3.3.1 Communication on RTEMS side

The following item describe the communication flow when a HRT needs to write/read data to/from a NRT (see dotted flow in figure 3.4).

- Write HRT $\rightarrow$ NRT. To write data to the memory buffer *rtems_to_lx*, the HRT needs to use the ISC API to call the *xluna_send_to_linux()* function, which will perform the requested action in 3 main steps:
  - Obtain *N_EMPTY* semaphore to check how many slots are available
  - Write on the message queue
  - Add an IRQ to signal Linux *N_FULL* semaphore. This is an example of the aforementioned IRQ usage (see 3.2).

- Read HRT ← NRT. To read data from the memory buffer *lx_to_rtems*, the HRT needs to call the API function *xluna_receive_from_linux()*. The body of this function is also divided in 3 main steps:
  - Obtain *N_FULL* semaphore
  - Read the data from the message queue
  - Add and interrupt request to the IRQ queue trough the IRQ manager. This request will increment the *N_EMPTY* semaphore (on Linux side), assuring that the data has been read.

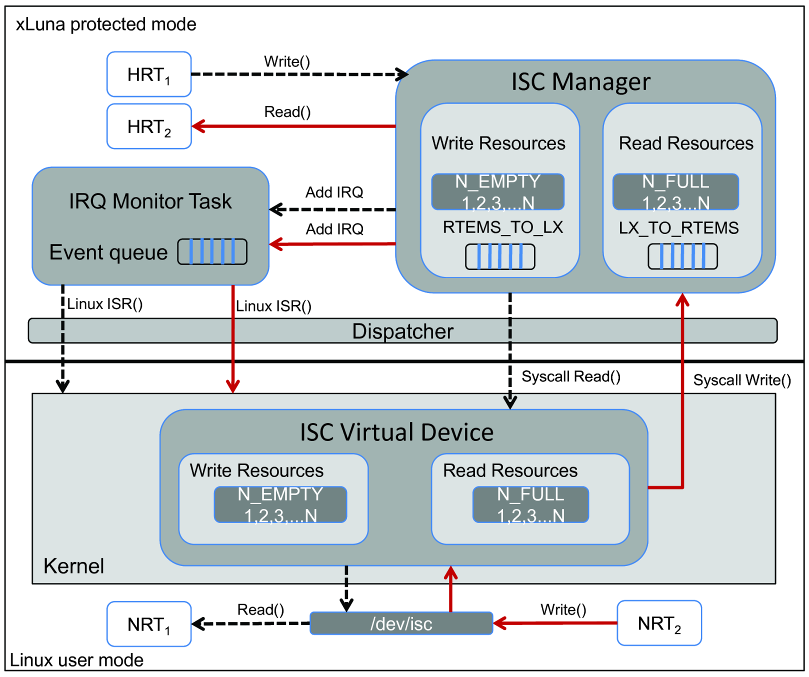


Figure 3.4: xLuna inter-systems communication

3.3.2 Communication on Linux side

The following item describe the communication flow when a NRT needs to write/read data to/from a HRT (see solid flow in Figure 3.4). To provide communication between the NRT and the ISC manager in RTEMS space a virtual device must be used (*isc_driver.c*). The ISC device, on time of initialization, creates the two global semaphores *n_empty* and *n_full*.

- Write NRT → HRT

- Decrement *N_EMPTY* semaphore (Linux side)
- Issue the system call to write the message and increment the *N_FULL* semaphore(see 3.4).

- Read NRT ← HRT
  - Decrement the *N_FULL* (Linux side).
  - Issue a system call to RTEMS for reading the data from the memory buffer and for incrementing the *N_EMPTY* semaphore (RTEMS side).

3.4 System call example

The ISC virtual device driver registers a virtual device in the Linux kernel and imports a device node (*/dev/isc*) for user processes to access. Through this virtual device, a Linux process can perform the usual file I/O operations. The actual read and write operations will perform communication with the peer ISC client inside the RTEMS subsystem. This device implements the functions needed to communicate with tasks in RTEMS space. Figure 3.5 shows a system call to write a message on RTEMS side.

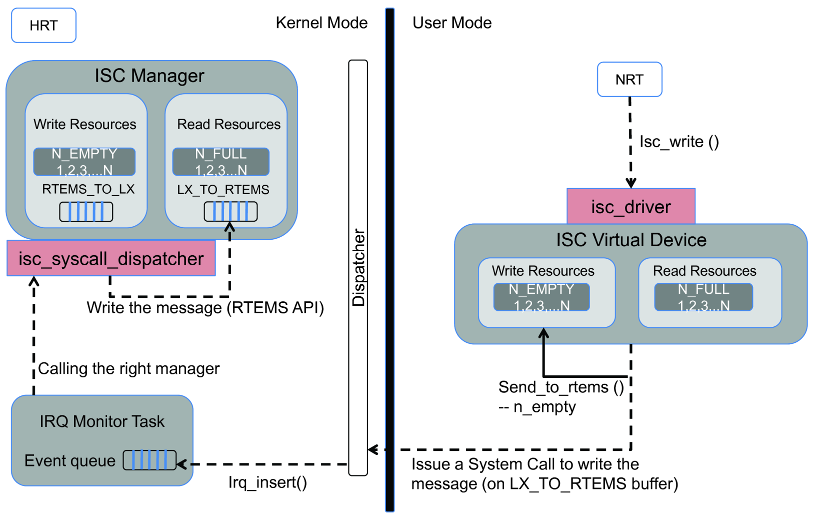


Figure 3.5: Write system call

The implementation ensures that the *N_EMPTY* semaphore on Linux side is decremented, to mark that one write resource was used by an NRT. Then a request to use the shared resources is made through a system call.

The *trap_dispatcher()* will insert an event on the IRQ event queue (IRQ Manager), this action will wake up the *irq_monitor_entry* task which was suspended until now. This task is responsible to treat the events inserted in the event queue. Each event will be dispatched to their handles according to their types (in this case *XLUNA_SYSCALL_ISC*) by calling the *irq_handle_event()* for each event on the IRQ event queue. This last function will send our request to the *irq_xluna_syscall_dispatcher()*, which finally will call the ISC Manager to make the requested of write the message on buffer. This aforementioned IRQ flow is the center target of the verification presented in this thesis.

3.5 Summary

This chapter presented an overview of xLuna architecture and its main modules. Some important details to understand our approach to verify the IRQ manager were also described. The next chapter shows the state of the art related to operating system verification.

Chapter 4

Operating System Verification: Related Work

The interest in formally verifying realistic and industrial low-level code and obtaining the highest standards of safety like, the Common Criteria EAL7 has grown significantly in recent years. This chapter presents a general overview of the work that is being made using formal methods in order to verify operating systems kernel.

The following sections are organized as follows:

- **Section 4.1** and **Section 4.2** give an overview on early and recent work related to operating system verification, respectively. For a more detailed presentation it is recommended the reading of Klein's article [20], which gives an excellent overview about work related to the subject.
- **Section 4.3** explains how the successor of Verisoft Project—Verisoft XT project—is using VCC tool to verify the Microsoft Hypervisor and PikeOS microkernel.

4.1 Early Projects

Proving software implementation correctness is a goal pursued for decades. The early work on formal verification of operating systems comes from Stanford Research Institute (SRI) international [21] in 1973-1980 with the **Provably Secure Operating System (PSOS)**. Neumann and Feiertag [22] gave a retrospective analysis of PSOS design—earlier documents in [23, 24].

PSOS goal was the construction of an operating system with provable security properties using an earlier example of hierarchical layered abstractions called Hierarchical Development Methodology (HDM) [24] which define a decomposition of development process with formal specifications defining each module and provides formal abstractions to module intercommunications. PSOS formal specifications were made using specification and assertion language (SPECIAL) [25].

Despite the original goal of reaching a fully formal verified operating system, PSOS ended with only some simple proofs demonstrated—"*some simple illustrative proofs were carried out, it would be a incorrect to say that PSOS was a proven secure operating system* "[22]. The HDM methodology and PSOS work were later used on SRI Prototype Verification System (PVS) [26] and Ford Aerospace Kernelized Secure Operating System (KSOS) [27, 28].

University of California at Los Angeles (UCLA) Data Secure Unix (DSU) was developed in the late seventies at the UCLA [29] and aimed to prove Operating System (OS) kernel implementation through program verification methods. Its architecture was similar to the modern microkernels providing threads, access control, virtual memory and input/output devices making these implementations the main effort of the verification work.

DSU kernel security proofs were divided in two steps, at the beginning were made bottom-up abstract specifications split in four levels [30], from Pascal kernel implementations, which had a clear formal semantics [31] and was considered to be suitable for low-level software, to top-level security properties. The second step was to prove that each level behavior was correct and consistent with each other.

The proofs were guided through XIVUS tool [32] based on first-order predicate calculus. Authors of [30] state that more than 90% of Kernel specifications were made and 20% of the code have been proven correct. Despite the verification proccess has shown some security bugs in earlier phases of the project, the authors conclude that "*the task is still too dificult and expensive for general use* "[30, Sect. 4].

Kernel for Isolated Tasks (KIT) is probably the first OS that deserves to be called full formally verified. Despite its size—620 assembly lines—the verification work has substantial meaning to the subject. As the name says, KIT addressed the problem of verifying properties for process isolation in a multi-tasking environment providing process scheduling and allocation of Central Processing Unit (CPU) time, response to program error conditions, massage passing between processes and I/O asynchronous devices [33, 34]. Nevertheless it did not provide support for dynamic process creation, file system or shared memory which is understandable considering its size.

KIT was not considered as a kernel for a general purpose operating system but more for small special purpose systems, which was also a limitation. The verification was done in the Boyer-Moore logic and conducted in the Boyer-Moore theorem prover [35], which was the predecessor of the ACL2 theorem prover [36]. KIT refinement-based verification methodology was actually very similar to UCLA Secure Unix, with a bottom-up abstract specification approach the state machine of the kernel running on hardware were the hardware machine states with memory, registers, flags and program counter. The abstract kernel in the middle defined the tasks scheduling and the communication primitives and the top-level abstract operation specifications defined the communication transitions [33]. These levels were interpreted as finite state machines and the correspondence proof, which is a state mapping from concrete operation to abstract operations, showed that on a single CPU the kernel implements the abstractions correctly.

KIT project proved task isolation, operating system protection from tasks, the inability of task entering in supervisor mode. In the process were revealed some some bugs such as naming incorrect registers or the bad restore of the task state after an I/O interrupt [34].

PSOS, DSU and KIT were pioneers in attempts to large scale software verifications and inspired some techniques still used nowadays. Despite the important advances made in these projects, formal verification of operating systems seemed at the time, too expensive, not suitable to realistic large systems (as is the case of KIT) or slow and incomplete verification work (like DSU and PSOS). Nevertheless the interest in operating systems verification returned after a decade. The next section gives an overview on more recent related work.

4.2 Recent Projects

The **Verified Fiasco (VFiasco)** project started in 2001 at Dresden University of technology [37] and aimed to formally verify C++ source code of the Fiasco kernel [38] which is an re-implementation of the L4 microkernel [39] with approximately 15K lines of code.

Hohmuth and Tews gave in [40] a summary of the VFiasco project where they, among other things, describe an experiment with the use of SPIN model checker [41] in an attempt to verify a reduced version(only two threads, no timeouts, no message buffers, no interrupts or page-faults) of Fiasco Inter-Process Communication (IPC).

The results from this work were not the most desirable due to the fact that SPIN extensively checks the whole state space, leading to the problem of state space explosion—"*the model checker used almost 2 GB main memory and more than 15 GB on the hard disk. We doubt that one can model check the full IPC path.* "[40, Sect. 2.3]. This approach was based on the construction of a model of the IPC using the Promela language [42] which then could be passed to SPIN model checker. A More detailed description can be found in [43].

Despite this experiment the project moved on with another approach and a subset of C++ was formalized using the PVS theorem prover [26], to reason about Fiasco implementations. This approach of verifying C++ source code was continued in **Robin Project** on the **Nova** Hypervisor [44] but neither VFiasco nor Robin could fully verify the kernel implementation. A more recent work related to the verification of Fiasco IPC was made by Schierboom in 2007 also using the PVS theorem prover [45].

Model checking approaches for verifying kernel implementation are also common practice. One of those examples is the **Fluke** IPC subsystem [46] where the SPIN model checker was used. Fluke IPC is the major and most complex component of the microkernel and it is highly concurrent. The authors state that the use of model checking instead of theorem provers implies a much lower learning curve. Yet, they also concluded that applying SPIN to the whole kernel would not be a good decision. Again, the state space explosion problem would not be desirable. Promela specification language and the SPIN model checker were also used to model and verify the multitasking multiprocessor operating system **RUBIS** [47]. In this work the IPC was also taken as a primary consideration. Once again a top-down approach to verification was used, high level abstract models and more detailed models of the RUBIS IPC were built. The more abstract levels were used to to prove properties like the absence of deadlocks an then, entering with more detail in lower level models, some aspects of the implementation and different IPC scenarios were taken under consideration and verified with SPIN. The work was worth it because it were found some important problems such as wrong return errors between tasks or not returning error messages at all. In this last case the task could not know whether an error had occurred or not which could lead to complete system failure. Another kernel which used SPIN as verification tool is the **HARMONY** portable real-time multitasking multiprocessor operating system developed at the Software Engineering Laboratory of the National Research Council of Canada. In [48] the approach used for Kernel formalization is similar to the RUBIS project. Different levels of abstraction models were created in Promela and checked with SPIN. The paper shows how IPC and task management were modeled and simulated with a significant number of different scenarios of tasks using different combinations of kernel services.

Coyotos secure microkernel-based operating system is the continuation of its predecessor EROS [49]. Coyotos is implemented in BitC [50], a language developed for the project with a precise formal semantics in order to perform verification of low-level Coyotos implementations. Starting from the fact that most of the kernel implementations were made based in unsafe languages like C, C++, and assembly, the idea of a kernel implemented in a safe language which facilitates the verification process was in fact appealing and the objectives for verification with this approach cover functional proofs of address translations and memory safety [51].

L4.Verified and Secure Embedded L4 (seL4). Klein et al [52] in 2005 argued about how the significant advances in theorem proving tools, all experiences in real industrial applications and at the same time, the increasingly choice for small but trusted OS kernels have stimulated the use of formal methods and consequent increase of software security and functionality. Due to this fact they decided to work on the verification of L4 microkernel.

L4 has suffered many re-implementations since its original version, among others [53, 54] in the aforementioned Fiasco project. The most recent work on L4 microkernel was carried out by the Australia's ICT Research Center of Excellence (NICTA)(NICTA) [55] in seL4 and L4. Verified projects, two projects that evolved closely related to each other with the primary objective of achieving an implementation correctness proof for seL4. L4.Verified starts after a first small pilot project that ended in 2004 to study the feasibility of a complete formal verification of the kernel, where basically there were three goals: (i) formalization of the L4 API using B Method where they were discovered ambiguities and inconsistencies in some places of the API implementation [52, Sect. 3], (ii) gaining experience to full verification of a small part of actual kernel code and (iii) developing a plan for full kernel verification.

After this pilot project the work on L4 verification continued on L4.Verified/seL4 projects in 2005 with two teams constantly providing feedback to each other. seL4 was concluded at the end of 2007 with a resulting small (8700 Lines of Code (LOC) C and 600 LOC assembly) microkernel to run in ARM architecture [56] and providing threads, IPC, virtual memory control, capabilities and interrupts control. The OS design team use Haskell [57] for building fast prototyping together with hardware simulator to be tested with user applications. Because Haskell is so related with Isabelle/HOL [58] the prototype specifications were translated to the theorem prover and served already as low-level formal designs useful for further efforts on L4.Verified work. Above C implementation the L4.Verified team built an model stack (similar to UCLA) that included the Isabelle/HOL output low/high-level design and an access-control model as the top layer [59] [20, Sect. 4.7]. For reasoning about the lowest layer with C and assembly implementation, the L4.Verified team had to model some hardware details like Memory Management Unit (MMU), exceptions and interrupts, then the next two design layers (low/high-level) contain the implementations of all important data structures with low-level details and the user-visible kernel operations respectively. The refinement proofs between abstract layers and executable specifications is 100K LOC of Isabelle proof.

In [60] Klein states, among other things, the microkernel verified functional correctness properties such as: no buffer overflows, no null pointers access, well formed structures, algorithmic invariants, etc. Among what they assumed to be correct, Klein shows the success and complete verification efforts in the scope of the project.

There are other projects which, although dealing with less complex kernels or not directly related with verification down to implementation level, made useful work related to the subject.

The **Flint** project is one of these efforts, Xinyu et al. [61] describe a way to formalize low-level code in an hardware interrupts and preemptive threads environment, which are constantly used in critical systems. Using ownership transfer techniques they show how disable/enable of interrupts in a concurrent setting can be formalized using a Hoare-logic style framework for reason about interrupt handlers, context switch and synchronization primitives.

The National Security Agency (NSA), Motorola and Kestrel Institute in 2000 designed and built together the **Mathematically Analyzed Separation Kernel (MASK)** to apply in Motorola cryptography smart card platform. The authors used Specware [62] to guarantee separation properties through several refinement proofs down to a close implementation level [63]. In the first phase of the project it were defined mathematical specifications for separation properties and multiple abstract levels specifying how MASK should enforce the separation specification. The last part of the project consisted in defining the data structures and algorithms used in the kernel. The separation kernel low-level design was translated into C code and analyzed against the Specware models.

Separation kernels are also often used in high assurance systems. Rockwell Collins and the U.S. Department of Defense presented in [64] a formal security policy for a separation kernel aimed for a Multiple Independent Levels of Security (MILS) architecture. A security policy is an important requirement for Separation Kernel Protection Profile (SKPP) [65] and is basically a formal specification of what is allowed in the system. The properties proved were based on previous defined theorems about *exfiltration*, *mediation* and *infiltration* [64, Sect. 3]. The authors use Common Lisp programming language to describe the security policy and the ACL2 [36] for proving the theorems expressed and the aforementioned security properties. The work also shows an example of a firewall which relies on the separation kernel security policy. Rockwell Collins **AAMP7** microprocessor also implements in hardware the properties of a separation kernel [66] in order to achieve Common Criteria EAL7 certification. The proof is achieved trough a high-level model that implements partitioning and a low-level design of the implementation micro processor code. The latter is translated manually and then executed in the ACL2 theorem prover.

Heitmeyer et al. [67, 68] at Washington Naval Research Laboratory [69] also provide formal verification for an embedded device that uses a separation kernel to ensure data separation but is unclear which device, kernel or Common Criteria evaluation level aimed. However the authors mention a small kernel (3000 LOC C) and less complex than a general purpose kernel. A Top Level Specification (TLS) was described in Timed Automata Modeling Environment (TAME) [70] with the guarantee that it is in agreement with the data separation security policy. As in Rockwell Collins approach, in order to have concordance with the security policy the kernel must implement a set of properties such as no-*exfiltration*, no-*infiltration*, temporal separation, separation of control and kernel integrity. The prove that the TLS enforces data separation the authors use PVS prover [26]. By correspondence proof between the TLS abstract states to Kernel concrete states and mapping assertion in the abstract TLS to assertion in the concrete code using Hoare-style pre/post conditions, the code conformance is demonstrated. The proof that the code was connect to the TLS was made manually. Other verification approach using TAME was also used in EAL4 certified **SELinux** [71].

The **Singularity** project at Microsoft Research started with the objective of designing a software platform with the main goal of dependability and answer to the question: "*what would a software platform look like if it was designed from scratch with the primary goal* *of dependability, instead of the more common goal of performance?* "[72, Sect. 1]. Hunt et al. show the Singularity project in [72, 73]. Singularity uses the advances in programming languages and tools to build new system architecture and a reliable operating system. The three key features on the Singularity architecture are (i) Software-Isolated Processes (SIPs) which can encapsulate an application or system providing failure isolation and strong interfaces, (ii) contract-based channels for fast and verifiable communication messages between processes and (iii) manifest-based programs for defining behavior properties of the code that runs on isolated processes. Approximately 90% of the system is written in Sing# [74] and C++ and assembly language. Sing# is an Spec# based type-safe, garbage-collecting programming language. Spec# [75] is an extension of the object-oriented language C# which porvides Hoare-style pre/post-conditions and object invariants for specification of program behaviour. Spec# static verifier uses Boogie [76] to generate logical verifcation conditions throught the annotated code and an automatic theorem prover that try to prove program correctness or find errors.

Some of the projects here spoken are research projects and despite the major contributions made in operating system verification they do not have the desired impact on the market or just were not built for this purpose. Nevertheless, there are some proprietary verified operating systems which are available in the market and are used in commercial and military planes, phones, cars, etc. One example of these systems is the Green Hills **Integrity®-178B** operating system [77]. Integrity is certified by the NSA-managed NIAP lab to EAL6+ High Robustness making it one of the most secure operating system. In order to be in accordance with the SKPP requirements and achieve such level of certification the kernel security policies, specifications and correspondence between design and implementation had to be formally proven, a complete test coverage of functional requirements and penetration tests by the NSA. The brand new Boeing 787 or the military Lockheed Martin F-35 Lightning II aircrafts are an example of Integrity kernel usage. Other commercial kernel under evaluation for EAL6+ is the Wind River **VxWorks** MILS platform [78] which already reached the SKPP requirements for medium robustness EAL4+. The BMW iDrive system and the Mars Reconnaissance Orbiter National Aeronautics and Space Administration (NASA) multipurpose spacecraft are two separate examples using VxWorks as its base system.

The following section describes the work made in Verisoft and VerisoftXT project. We give special attention to the VerisoftXT project due to the fact that it is tightly connected to the VCC development and usage.

4.3 Verisoft/VerisoftXT Project

Verisoft [79] and VerisoftXT [80] projects are an example of how years of research can be placed in practice and used in real-world applications. The Verisoft project started in 2003 and it was scheduled to end in 2007 with the objective of providing a pervasive[1] formal verification of a complete operating system stack from hardware to user applications. From bottom to top:

- Hardware
  - The **VAMP** microprocessor in the lower-level gives a formal consistent foundation for the upper levels [81, 82]. The PVS theorem prover was used to formally verify the instruction set behavior.
- Kernel Mode
  - On top of hardware level is the **Communicating Virtual Machines (CVM)** defining the hardware dependent code of the kernel, such as device drivers and memory paging mechanism, and the hardware independent interface [83]. This separation between hardware dependent and independent parts enabled the isolation of assembly code which was positive for the verification work.
  - The **VAMOS** layer defines the code running in kernel mode and together with the CVM layer form the OS kernel [84]. The verification is not yet completed.
- User Mode
  - The next layer implements the user-mode **Simple Operating System (SOS)** [85]. It provides file based I/O, IPC, sockets and procedure calls interface for applications. The verification work in this level was also incomplete.
  - The last layer resides the user applications. In [86] is shown an example of a formally verified email client.

[1]Do not rely on compiler correctness nor instruction set model and formally verify all these steps to form a complete formal chain

The following text describe two projects within VerisoftXT, the (i) Subproject Avionics where the objective is to prove functional correctness of the SYSGO PikeOS microkernel-based partitioning hypervisor and the (ii) formal verification of the Microsoft Hyper-V Hypervisor platform. The Verisoft successor began in 2007 as collaborative research project between Germen Research Center for Artificial Intelligence (DFKI), Microsoft Research,EMIC and Saarland University. It is scheduled to end in 2010.

Microsoft Hyper-V Hypervisor is a 100K LOC C and 5K LOC assembly software layer that runs on x64 hardware and has the ability to turn a x64 multi-processor with virtualization extensions[2] into a number of virtual x64 multi-processors. Leinenbach and Santen show in [89] that applying formal methods to a system like Hyper-V, which was not implemented with verification in mind, has many challenges. The authors state that in the verification process developers and testers shall maintain the annotations together with the code, which fits perfectly into VCC's methodology. The Hypervisor is mostly implemented in C, and as C has an unsafe type system where memory allocation or deallocation is made explicitly so memory safety needs to be verified. VCC also solves this problem using a typed view of the memory based on Spec#-style object ownership methodology, the VCC memory model [14] provides means to have a typed view of the objects in memory and reason about its *validity* (see Chapter 2). Another challenge has to do with the fact that the Hypervisor is a concurrent system and the original code shall not be modified to facilitate the verification task. As previously stated (Chapter 2) VCC is suitable for concurrency verification.

Correctness of concurrent code is, in most cases, achieved by proving a simulation theorem between the code and an abstract model of the system. For the Hypervisor verification a VCC *ghost* top-level model for each partition (guest system) of the Hypervisor was built to take information about IPC, privileges, a set of x64 processor states, etc. The development of the x64 architecture top-level model has been carried out by the Saarland University and is also being used in proofs for the low-level assembly code [90]. As the authors explain in [89, Sect. 3], build an abstract model of the underling processor cannot be made deterministically because other core-extern events (memory changed by other cores, interrupt requests from devices or other cores,etc) can happen non-deterministically. The verification team together with VCC solves this problem by adding a two-state invariants in the abstract model for specifying the correct and legal behavior of each state transition. Connecting invariants within concrete objects and invariants in the abstract model VCC

[2]AMD SVM [87] or Intel VMX [88] virtualization extensions

will ideally prove the *admissible* actions and the correct implementation of the Hyper-V concerning the abstract model.

VCC tools has been successfully applied and certainly improved in this project. In [7], some results of the verification work are shown. It were inserted into the code-base 13500 lines of VCC annotations, 350 functions have been successfully verified as well as invariants for about 150 C structures have been verified and prove *admissible*. Despite the fact that "*The Hypervisor is part of a released product with very low defect density.*"and "*indeed less than a handful have been found during the verification process* "[89, Sect. 3]. The authors also concluded that the usage of this verification methodology and tools since the beginning would have improved the quality and prevent defects.

Another work within VerisofXT is the Subproject Avionics where the objective is to prove functional correctness of the SYSGO **PikeOS** microkernel developed to the DO-178B avionics safety standard and aimed at Common Criteria EAL7 [91]. As in Hyper-V, PikeOS provides partitions virtualization for guest operating systems running on the same CPU. It can thus be used as a paravirtualizing hypervisor for several architectures (x86, PowerPC, ARM and others) for developing embedded systems in a secure environment with real-time capabilities, kernel resources and thread communication mechanisms (shared memory, IPC and events).

The verification approach in this work is also conducted by VCC Tools and methodology. Baumann et al. [92] show the first verification results relating functional properties. The first parts of the code being verified were chosen by their small level of dependence with other parts of the kernel. The next target was the verification of the system calls whose execution spans through all levels of the microkernel, thus creating a much more complex scenario to verification. One system call to change the priority of a thread is given as an example in [93]. System call verification involves reasoning about the underling hardware and assembly parts of the implementation code. To address this problem the relevant parts of the hardware were modeled as C structures using the VCC capabilities to create ghost C code only for verification purposes. This model enabled the verification of low-level functions. For example, assembly PowerPC (the chosen target platform for verification) instructions are implemented in ghost state and verified in VCC as normal (ghost) C functions. Specifications were also added into kernel C code and a more general abstract model of the kernel functionalities was built in ghost state and connected with the real code trough object invariants. At the end this approach did provide a complete structure for machine-checked proof [94].

For performance and real-time reasons there are preemptive parts of the microkernel which force reasoning about concurrency and switching problems. In order to guarantee non-interference accessing shared physical hardware the verification team enforce the access policy to shared resource (locking) verifying PikeOS lock implementation and thus ensuring exclusive access to the running thread [95]. The lock verification approach to shared access is inspired in [96] and is also used in this thesis. Context switch problems were also address, using the VCC PowerPC model to reason about system state and load/save context when interrupts involve switching. (For more details please refer to [95, Sect. 8].)

PikeOS verification is still ongoing work. Despite the major verification advances and improvements to verification methodology and structure there are some kernel capabilities that have not yet been addressed (e.g. IPC). In [95] the authors conclude that a possible future work in PikeOS formal verification is to prove correctness of the PikeOS system instead of PikeOS microkernel alone, thus providing a formally verified hypervisor.

4.4 Summary

Despite formal verification of low-level operating systems code was initially seen as an prohibitively or even impossible task due to the size and complexity of the problem, this view has changed over time and projects aiming to use of formal methods in large low-level systems grew positively in the last years. Achieving the highest security Common Criteria requirements is now a target for both industrial and research areas. Common Criteria defines 7 levels of assurance and the last 3 levels requires the use of formal or semi-formal methods. The EAL7 level requires the use of formal methods in requirements, functional specification, high-level design and semi-formal for low-level design.

For the best of our knowledge only Integrity as reached the EAL6+ level. In the projects listed here, one can see that different formalisms such as model checking, automated first-order theorem provers or interactive methods were used. Yet, as showed in the VerisoftXT project the trend is to drive the verification through more automated tools.

The verification tool chosen to be applied in xLuna was also VCC. The following chapter shows the results of that work.

Chapter 5

IRQ Manager Formal Verification

In this chapter we give a detailed description of the modular verification approach applied to the xLuna IRQ module. We expose the *contract* embedded into each IRQ function as well as the methodology followed for both path, functional and safety requirements, and concurrent verification.

When dealing with a multi task environment, the system has to process several events through interrupts with associated handlers. Thus, the main objective of this xLuna module is the correct dispatching of such events. When Linux task is running and an interrupt/trap occurs a xLuna dispatcher function saves it (see Figure 5.1). When the IRQ manager starts or an event is saved, another task is resumed for treat the event inserted (see Figures 5.35.4).

Our verification target is the correct event treatment process.

5.1 Verification Target

Verifying low-level code that was not implemented with formal verification in mind and adapting the verification methodology to that implementation is a non-trivial task. When reasoning about xLuna as a formal verification target its own architecture suggests that a modular approach should be taken in order to achieve an overall correctness proof. One of the most critical and crucial modules of xLuna is the IRQ manager, since it has to catch and process all interrupt requests needed for proper system functionality. It thus constitute and interesting target for formal verification and was the subject of our study. Yet, the different modules are considerably dependent on each other. Moreover, kernel code is highly dependent of machine assembly instructions, which causes issues for the verification task since VCC does not interpret assembly language. For this reason, it was decided, at this stage, to assume machine instructions and inlined assembly correct. All IRQ module dependencies were studied to build proper code isolations and abstractions were made to fit the verification methodology.

5.2 IRQ Design

Following xLuna architecture (see Chapter 3), the Linux kernel is running as an unprivileged (user-mode) RTEMS task and thus, it does not have direct hardware access. The main purpose of the IRQ manager is to serve as a bridge connecting the Linux subsystem to hardware interrupts. Hardware interrupts or software traps are both called *events* that are inserted into an *Event Queue* to be sent to their handlers.

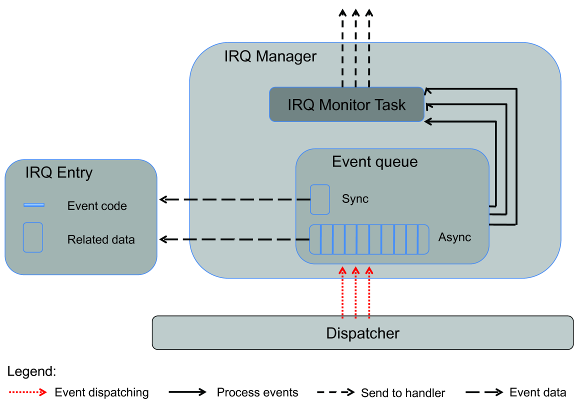


Figure 5.1: Interrupt request manager

Interrupts are filtered by a dispatcher function which is responsible to insert them into the event queue. Once there are events in the queue they should be processed by the IRQ manager through a *monitor task* and for each event call the respective Linux handles or xLuna services (see Figure 5.1). Events can be treated synchronously or asynchronously. In the queue structure there can be only one *sync* event at a time. This event is a request caused by the running instruction and must be processed synchronously. *Async* events are accumulated in the queue according to their event type (e.g., TT_ISC_RTEMS_TO_LX is the special trap type 0x21 for inter systems communication that is inserted as an async event). As shown in Figure 5.1, both events have an associated data structure which contains the data needed for event handlers and the the interrupt/trap code.

In the next sections, we describe the approach taken for verifying the treatment of each event inserted in the queue and the correct usage of data structures involved in this flow.

5.3 VCC Verification Approach

We can drive the verification methodology used through an IRQ manager function used to insert a synchronous event into the queue. As already mentioned, VCC enforces a ownership model to guarantee a consistent and typed view of all objects (e.g., pointers to data structures) which ensures that objects of the same type and different addresses do not overlap in memory. [14].

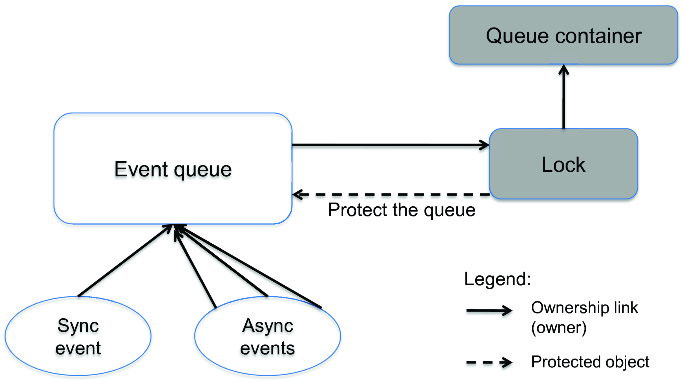


Figure 5.2: VCC event queue ownership

Figure 5.2 shows an high-level view of our ownership setup. Basically, the figure says that the event queue will be the owner of all synchronous and asynchronous events entering in IRQ manager. The Lock (which is a spin lock to ensure mutual exclusion) protects and is the queue structure owner. Moreover, the Lock is himself owned by the container object. Nevertheless, more details about configuration of this setup are given throughout this chapter.

When xLuna interrupt manager is initialized VCC sees all objects as mutable. Therefore ownership relations must to be configured at the program entry point (see Listing 5.6 for *irq_init()* function) to prevent further object updates without complying with the ownership protocol.

After initialization function, the IRQ manager will start the events dispatching and the execution flow spans throughout other functions. Figure 5.3 shows the IRQ manager simplified Call Graph of the target functions for verification in this section.

Verification of RTEMS code is out of scope of this work. Therefore, RTEMS API calls were assumed pure for the verification.

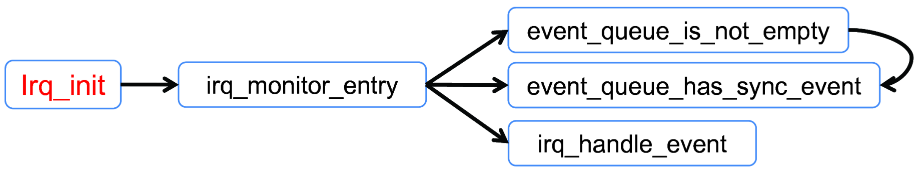


Figure 5.3: Call Graph for *irq_init()* function

5.3.1 Queue types and invariants

In this section we show the queue structures/objects for real code (in Figure 5.2) as well as its invariant that are supposed to hold according to the VCC ownership protocol. Asynchronous events have to stay within bounds and must have no associated *data* (invariant in Line 4). When is a sync event, *data* must be non NULL and well typed in memory.

```
1 typedef struct irq_entry {
2   unsigned32 event;
3   void *data;
4   invariant(data == NULL <==> (event>=0x10 || event<=0x22))
5   invariant(data != NULL <==> typed(data))
6 }irq_entry;
```

Listing 5.1: IRQ entry event

The queue invariants guarantee that the queue is the owner of all *event* and that the number of *pending event* will be positive.

```
struct event_queue_entry {
  struct irq_entry sync_event;
  struct {
    unsigned int pending;
  } asyn_events[NUMBER_OF_ASYN_EVENTS];
  invariant(keeps(&sync_event))
  invariant(forall(unsigned i; i < NUMBER_OF_ASYN_EVENTS; keeps(
      asyn_events+i)))
  invariant(forall(unsigned i; i>=0 && i < NUMBER_OF_ASYN_EVENTS;
      asyn_events[i].pending >= 0))
}event_queue;
```

Listing 5.2: IRQ queue

5.3.2 VCC ghost types, invariants and initialization

In this section we show the VCC ghost objects only for verification purposes (surrounded with spec(...)). This is related to the gray part of Figure 5.2. This objects are used for concurrency verification, but the Lock and Container objects are initialized and configured at IRQ manager boot time.

```
1 spec(
2 struct vcc(claimable) _QUEUE_CONTAINER {
3 int dummy;
4   invariant(keeps(&QueueLock))
5   invariant(QueueLock.protected_obj==&event_queue)
6 } QueueContainer;
7 )
```

Listing 5.3: Queue container object

In Line 4 and 5, invariants say that the $QueueContainer$ will allays keep $QueueLock$ in its ownership set and that the $QueueLock\ protected\_object$ is the queue.

```
1 spec(
2 typedef struct vcc(volatile_owns) _LOCK {
3   volatile int interrupt_disable;
4   obj_t protected_obj;
5   invariant(interrupt_disable == 0 ==> keeps(protected_obj))
6 } LOCK;
7 )
```

Listing 5.4: Ghost lock object

The lock object simulates whenever the interrupts are enable or disable. In the real code the enabling/disabling interrupts are done trough assembly atomic operation when calling $sparc\_disable\_interrupts()$ or $sparc\_enable\_interrupts()$ functions. The object declaration in Line 4 points to the queue (from $QueueContainer$ invariant) and Line 5 says that whenever the interrupts are enable (zero) the lock is the owner of the queue.

5.3.3 Verification of *lock_init()* function

```
speconly(
void lock_init(LOCK ^l spec(obj_t queue))
  writes(span(l),queue)
```

```
 4     requires(wrapped(queue))
 5     ensures(closed(l) && l->protected_obj == queue)
 6     ensures(closed(queue))
 7 {
 8     l->interrupt_disable = 0;
 9     speconly(
10        l->protected_obj = queue;
11        set_owns(l, SET(queue));
12     wrap(l);
13     )
14 }
15 )
```

Listing 5.5: VCC ghost function for lock initialization

Listing 5.5 shows the lock initialization. It must ensure that after the initialization the the protected object is the queue (Line 5), the both objects are closed to assert its invariant (Lines 5 and 6) and the lock owns the queue (Line 11).

5.3.4 Verification of *irq_init()* function

The *irq_init()* function is the main entry point of IRQ manager. It is also the function where we should tell VCC what is our ownership configuration.

```
rtems_status_code irq_init(void)
 writes(set_universe())
 requires(program_entry_point())
 ensures(closed(&event_queue))
 ensures(linux_interrupt_enabled==1 &&
     linux_hardware_handler_in_progress==0)
{
   event_queue_reset(); //Queue allocation
 spec(claim_t c;)
 spec(
   unsigned j;
   for (j = 0; j < NUMBER_OF_ASYN_EVENTS; j++)
     invariant(j <= NUMBER_OF_ASYN_EVENTS)
     invariant(forall(unsigned i; i < NUMBER_OF_ASYN_EVENTS; i < j ?
       wrapped(&event_queue.asyn_events+i) : mutable(&event_queue.
          asyn_events+i)))
     writes(array_range(&event_queue.asyn_events,NUMBER_OF_ASYN_EVENTS))
16       {
17         wrap(&event_queue.asyn_events+j);
18       }
19   set_owns(&event_queue,array_members(&event_queue.asyn_events,
         NUMBER_OF_ASYN_EVENTS));
20   set_owner(&event_queue.sync_event,&event_queue);
21   wrap(&event_queue.sync_event);
22   wrap(&event_queue);
23   assert(wrapped(&event_queue));
24   assert(&event_queue == owner(&event_queue.sync_event));
25   assert(&event_queue == owner(&event_queue.asyn_events[0]));
26   ) //End spec
27     //assumed(true) rtems monitor criation
28     assert(wrapped(&event_queue));
29     spec(lock_init(spec(&QueueLock) spec(&event_queue));)
30     assert(closed(&event_queue));
31     set_owner(&QueueLock, &QueueContainer);
32     wrap(&QueueContainer);
33 
34     speconly(
35         c = claim(&QueueContainer, closed(&QueueContainer));)
36       // start monitor
37       irq_monitor_entry(0 spec(c));
38 
39     assert(closed(&event_queue));
40 
41     linux_interrupt_enabled = 1;
42     linux_hardware_handler_in_progress = 0;
43     return RTEMS_SUCCESSFUL;
44 }
```

Listing 5.6: IRQ Manager Initialization Function

The first two annotations state that this is the initial function. In Line 2 we tell VCC that all global objects are considered to be mutable at this point, therefore, we want to have access to them and manage its behavior from now. The program entry point state is marked by the *program_entry_point()* function in Line 3 which implicitly takes state as parameter.

Now we need to set the ownership relations for all *async* objects in the queue. Currently VCC can not wrap all objects of an array in one call. Therefore, in order to configure ownership relations for *async* events we have to do it iteratively through a ghost loop (Line

11 to 18). The loop invariant says that if $i < j$ the *async* event i is already wrapped (owned by the current thread). Otherwise, the *async* event i is mutable (Line 13 and 14). This holds after each iteration. We also state that the loop will only operate within array bounds (Line 12).

Write permissions to change the array also have to be guaranteed as a loop invariant (Line 15). The loop body then wraps the object (Line 17).

We can now say that the queue is the owner of all *async* objects because they are all wrapped (Line 19). One can only close the owner object if all its embedded objects are wrapped. *Sync* events are also owned by the queue (Line 20).

Now we can wrap the queue inside out and assert its ownership properties (Lines 21 to 25). According to the setup shown in Figure 5.2, after IRQ manager boot we ensure that:

- The queue is *typed* and protected at the end of initialization (Line 4);
- The queue is the owner of all *sync* and *async* events (Line 24 and 25);
- When the queue is *closed* its invariant and the invariants of all embedded types hold (see Subsection 5.3.1).

Line 29 initializes the lock (see Listing 5.5) and in Line 31 we set the $owns$ field of $QueueContainer$ (container now owns the lock). The container invariant holds when wrapping it at Line 32.

Now we can create a claim on the container and pass it as ghost parameter to other function (Line 37). The claim ensures that the $QueueContainer$ will remain closed. That knowledge will be useful for verification in other functions.

5.3.5 Verification of *irq_monitor_entry()* function

```
1 static rtems_task irq_monitor_entry(rtems_task_argument ignored  claimp(
     c))
2 writes(&sync_events_handled)
3 writes(&linux_isf)
4 writes(&linux_hardware_handler_in_progress)
5 reads(&linux_interrupt_enabled)
6 requires(thread_local(&linux_interrupt_enabled))
7 { irq_entry irq;
8   do { while(event_queue_is_not_empty()) {
9         if (event_queue_has_sync_event()) {
```

```
10          ASSERT(0==event_queue_pop(&irq spec(c)));
11          unchecked(sync_events_handled++);
12          irq_handle_event(irq.event, irq.data);
13          }
14        else if (LINUX_INTERRUPT_ENABLED){
15            ASSERT(0==event_queue_pop(&irq));
16            irq_handle_event(irq.event, irq.data);
17          }
18          else {
19            rtems_task_suspend(RTEMS_SELF);
20          }
21      } //End While
22      rtems_task_suspend(RTEMS_SELF);
23   } while(1); //while true
24 }
```

Listing 5.7: IRQ monitor task

This function is responsible to pop the *events* and send them to their handlers. Basically the function is a loop iterating over the queue, but the actual work is done in the called functions. For the monitor function we need to frame the access to manager state flags and ensures the queue good health for the necessary called functions.

- Write and read access to state flags are framed in Lines 2 to 5. Write access implicitly assert thread locality. However, the monitor will only needs to read $linux\_interrupt\_enabled$, so we state a precondition saying that $linux\_interrupt\_enabled$ is local to the current thread (Line 6).

- In Line 12 VCC complains about a possible overflow. This can properly avoided by the $unchecked()$ clause. Even so, limits shall be bounded in the real code to avoid overflow.

- Note that when the $event\_queue\_pop()$ is called (Lines 10 and 15) the ghost parameter takes the necessary knowledge with it. This will be necessary to prove that the $event$ was removed with non-interference.

5.3.6 Verification of *event_queue_is_not_empty()* function

```
 1 static inline unsigned32 event_queue_is_not_empty(void)
 2 requires (thread_local(&event_queue.sync_event))
 3 requires (thread_local(as_array(&event_queue.asyn_events,
     NUMBER_OF_ASYN_EVENTS)))
 4 reads(span(&event_queue))
 5 ensures(result == 1 ==> (&event_queue.sync_event != NULL)||(&event_queue
     .asyn_events != NULL))
 6 {
 7   int i;
 8   if (event_queue_has_sync_event()) return 1;
 9
10   for(i = 0; i < NUMBER_OF_ASYN_EVENTS; i++)
11     invariant(0 <= i <= NUMBER_OF_ASYN_EVENTS)
12     invariant(event_queue.asyn_events[i].pending >=0)
13     invariant(forall(unsigned i ; 0 <= i < NUMBER_OF_ASYN_EVENTS; typed
         (&event_queue.asyn_events[i])))
14   {
15     if (event_queue.asyn_events[i].pending) return 1;
16   }
17   return 0;
18 }
```

Listing 5.8: Event queue is not empty function

This function will not change the queue, it will just see if there is events and inform that trough the return values. In this case we only need to prove that is running in the current thread domain (Lines 2 an 3) and only with read permissions (Line 4).

- If the queue have a $sync$ or $asyn$ event it will return 1. The postcondition in Line 5 will ensure that the event is non null.

- For $asyn$ event a loop is done to search pending events. The loop invariants in Lines 11,12 and 13 will ensure respectively that:

  - The loop will iterate within bounds
  - That pending is positive
  - That all $asyn$ are well typed in memory.

5.3.7 Verification of *event_queue_has_sync_event()* function

```
1 static inline unsigned32 event_queue_has_sync_event(void)
2 requires (thread_local(&event_queue.sync_event.event))
3 reads(&event_queue.sync_event.event)
4 ensures(result != 0 ==> &event_queue.sync_event.data != NULL)
5 {
6   return event_queue.sync_event.event != 0;
7 }
```

Listing 5.9: Event queue has sync event function

The Event queue has sync event function will also need permissions to read the queue (Line 2 and 3). Remember that the queue will only accept one *sync* event at the time and *sync* events have always *data* associated. Therefore, if the result of the return value is zero different (*sync* event do not exists) one need to make sure that the event queue data is non null (postcondition in Line 4).

5.3.8 Verification of *irq_handle_event()* function

Function responsible for event dispatching.

```
static void irq_handle_event(unsigned32 event, void *data)
requires(event == TT_UART_FIRST || event == TT_TIMER ||
      event == TT_ASYN_CLEAR_WINDOW || event == TT_ASYN_CLEAR_WINDOW ||
      event == TT_ISC_RTEMS_TO_LX || event == TT_ISC_LX_TO_RTEMS ||
      event == TT_LINUX_SYSCALL ||event == TT_SYNC_KILL_USER ||
      event == TT_SYNC_CLEAR_WINDOW ||event == TT_MNA ||
      event == TT_FPD ||event == TT_DATA_ACCESS ||
      event == TT_INST_ACCESS||event == TT_XLUNA_SYSCALL ||
      event == TT_UNHANDLED_EXCEPTION)
requires(thread_local(data))
writes(&linux_hardware_handler_in_progress)
writes(&linux_isf)
{
  switch(event) {
    case TT_UART_FIRST:
    case TT_TIMER:
    case TT_ASYN_CLEAR_WINDOW:
    case TT_ISC_RTEMS_TO_LX:
    case TT_ISC_LX_TO_RTEMS:
```

```
20        linux_hardware_handler_in_progress = 1;
21        ASSERT(linux_interrupt_enabled);
22      case TT_LINUX_SYSCALL:
23      case TT_SYNC_KILL_USER:
24      case TT_SYNC_CLEAR_WINDOW:
25      case TT_MNA:
26      case TT_FPD:
27      case TT_DATA_ACCESS:
28      case TT_INST_ACCESS:
29        ASSERT(linux_isf);
30        irq_build_linux_frame(event, linux_isf);
31        linux_isf = NULL;
32        rtems_task_suspend(RTEMS_SELF);
33        break;
34      case TT_XLUNA_SYSCALL:
35        irq_xluna_syscall_dispatcher(data);
36        break;
37      case TT_UNHANDLED_EXCEPTION:
38        irq_do_unhandled_exception(data);
39        break;
40    }
41  }
```

Listing 5.10: Handle event function

- Line 2. The input trap/event type must be known.
- Line 10. $data$ shall be local to the running thread.
- Line 11 and 12. Write permissions for the addresses updated in this function.

5.4 Concurrency Verification

The above described approach is suitable for sequential code. Nevertheless, the kernel has to guarantee that executions are made without interference or unexpected kernel behavior. In a concurrent real-time kernel, not only user processes are preemptable but also kernel processes may be subject to stringent scheduling policies or interrupts.

In order to achieve non-interference updates to critical regions in xLuna, a low-level function is called to restrict concurrent access by disabling interrupts, performing the required update steps, and then enabling interrupts again. Since this function is implemented in assembly language, in VCC we would have to analyze its behavior using a ghost assembly model (as stated in Section 4.3 for Microsoft Hyper-V Hypervisor [90]).

5.4.1 Lock approach

Disable interrupts gives to the running thread non-interference execution steps before interrupts are enabled again. One can think about this low-level access policy as a mandatory lock and use an abstract ghost implementation suitable for VCC methodology (see for instance Listing 5.4). When interrupts are disabled we call a VCC ghost specification function (see Listing 5.16 Line 16) that will transfer ownership domain to the running thread and prevent preemption while interrupts are disable (see for instance Listing 5.16 line 26).

This allows reasoning about implementation steps in a sequential fashion. Still, one needs to ensures that the lock is not destroyed or deallocated: in VCC we can model this issue through a $claim$. The gray part of Figure 5.2 shows the ownership configuration for ghost code and states that the $QueueContainer$ ghost structure is the owner of the lock and the latter owns the queue. The container has two invariants saying that (i) the object protected by the lock is the queue and (ii) enforcing the fact that it is the owner of the lock (see Subsection 5.3.2).

Figure 5.4 shows the call graph of the functions explained in this section.

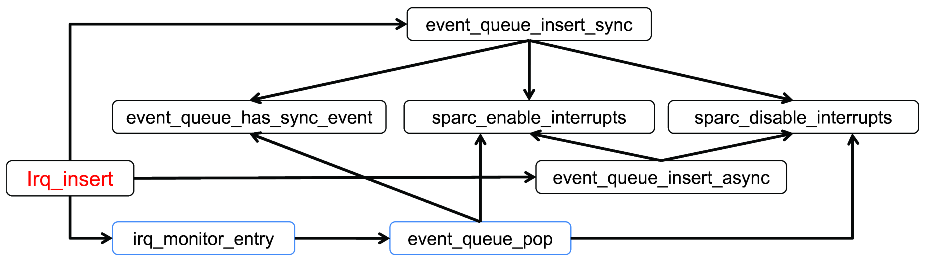


Figure 5.4: Call Graph for *irq_insert()* function

5.4.2 Verification of *irq_insert()* function

VCC enables the implementation of *pure* functions to be used inside other VCC clauses (requires, ensures, assert etc.). This functions are always marked with an *ispure* at the beginning of its declaration and they are not required to have a body. In xLuna *irq_insert* the above *pure* function *VCC_SPEC_FUN_is_asyn_trap* is used to verify if the trap is indeed asynchronous or synchronous when it is denied.

```
spec(
  ispure bool VCC_SPEC_FUN_is_asyn_trap(unsigned32 trap)
    returns(((trap)>=TT_ASYN_MIN) && ((trap)<=TT_ASYN_MAX));
)
```

Listing 5.11: Is asynchronous trap VCC specification function

Now we explain the contracts added to *irq_insert* function.

```
rtems_status_code irq_insert(unsigned32 event, void *data, unsigned32
    sync claimp(c))
 reads(event,data,sync)
 maintains(mutable(&irq_monitor_id))
 requires((sync==ASYN_EVENT)<==> VCC_SPEC_FUN_is_asyn_trap(event) && (
     data ==NULL))
 requires((sync==SYNC_EVENT)<==> !VCC_SPEC_FUN_is_asyn_trap(event) && (
     data !=NULL))
 ensures(result == RTEMS_SUCCESSFUL || result == RTEMS_TOO_MANY)
{
  irq_entry irq;
  int ret;
  static int queue_full_count;
  irq.event = event;
  irq.data = data;
  if (sync == SYNC_EVENT) {
    assert(sync == SYNC_EVENT);
    assert(irq.data != NULL);
    assert(!VCC_SPEC_FUN_is_asyn_trap(irq.event));
    if ( 1 == (ret=event_queue_insert_sync(&irq spec(c)))) {
      ASSERT(0);
    }
  } else {
    ASSERT(sync == ASYN_EVENT);
    assert(sync == ASYN_EVENT);
    assert(irq.data == NULL);
24      assert(VCC_SPEC_FUN_is_asyn_trap(irq.event));
25      if ( 1 == (ret=event_queue_insert_asyn(&irq spec(c)))) {
26        if (++queue_full_count > 100) {
27          ASSERT(0);
28        }
29      } else queue_full_count = 0;
30    }
31    rtems_task_resume(irq_monitor_id);
32    if(ret==0) return RTEMS_SUCCESSFUL;
33    else return RTEMS_TOO_MANY;
34  }
```

Listing 5.12: IRQ insert function

The *irq_insert* function is used to insert a new event in the queue. It receives as input parameters an event number, a pointer to data and and one *sync* variable which will tell if the event is *sync* or *asyn*. In Line 2 we establish the read permissions for this three input parameters.

After each event inserted, the *irq_insert* will resume the monitor task. The monitor task has an associated global pointer which has to be mutable. Otherwise VCC will complain about the permissions access this pointer. This requirement is stated in Line 3.

To reason about the *sync* parameter we use the *pure* function explained above (in Listing 5.11). Therefore, as input requirement we say that whatever is the type of the *sync* variable ($ASYN\_EVENT$ or $SYNC\_EVENT$) it is always a true event.

- Lines 14 and 16 *assert* the *sync* parameter and *event* again.
- Line 15 guarantees the data validity.

xLuna has its own $ASSERT$ function. Line 22 is replicated to check the *asyn* variable in VCC. We also *assert data* and *event* validity for *asyn* events. For last, the postcondition in Line 6 ensures the two possible outcomes from this function.

5.4.3 Verification of *sparc_disable_interrupts()* function

Synchronization is needed to ensure mutual exclusion or the absence of race conditions in concurrent programs. For this purpose low-level atomic operations are often used. xLuna ensures non interference operations by disabling/enabling interrupts using assembly functions for the SPARC processor. The SPARC processor provides atomic operations like *ldstub* (load and store unsigned byte) and *swap* (see [97]). These operations are atomic exchanges in a special CPU register.

In the scenario when two threads are attempting to disable interrupts (acquiring the lock, in our model), one of the threads has to do it first since we are reasoning about atomic instructions in a single processor. Therefore, our lock abstraction needs a way for reason about atomic low-level operations.

```

vcc(atomic_inline) int SPARC_test_and_set(volatile int ^Dest  ,int Exch
    ,int Comp) {
  if (*Dest == Comp) {
    *Dest = Exch;
    return Comp;
  } else {  return *Dest; }
}
```

Listing 5.13: VCC function for simulate atomic SPARC operations

In VCC we can model low-level atomic operations using functions marked with *vcc*(*atomic_inline*). Such functions are interpreted as a simulation of the underlying hardware atomic operations and can only be called inside VCC atomic blocks (as in Function 5.14), counting as a single update.

The above function is a compare-exchange which checks if $Dest == comp$ and store $Exch$ in $Dest$ if they are equal (as a SPARC atomic *ldstub*). This means that the lock was acquired, or in our abstraction, interrupts are disabled.

The simulation function that replaces (for verification) the original *sparc_disable_interrupts* assembly function is given bellow.

```
speconly(
void VCC_SPEC_FUN_sparc_disable_interrupts(LOCK ^l claimp(c))
  always(c, closed(l))
  ensures( wrapped(l->protected_obj) && is_fresh(l->protected_obj))
{
  int acquired = 0;
  do {
    atomic (c, l) {
      acquired = SPARC_test_and_set(&l->interrupt_disable , 1,0) == 0;
      speconly(if (acquired) giveup_closed_owner(l->protected_obj , l);)
    }
  } while (!acquired);}
)
```

Listing 5.14: VCC function for simulate disabling interrupts

To change the event queue without interference, the current thread has to acquire the lock (disable interrupts) calling this specification function.

- The *claim* ensures that the lock will remain closed and thus, can not be destroyed (Line 3).

- The VCC atomic block enables only one atomic operation, which is our SAPAC atomic function (Line 9) and multiple specification operations. Nevertheless, we only need to transfer the ownership domain of the queue ($l \rightarrow protected\_ogj$) from the lock ($l$) to the current thread (Line 10). Exclusive access to the queue is thus guaranteed until interrupts are enabled again.

- The is_fresh annotation ensures that the queue was not owned by other thread before. Between disable/enable interrupts the is is always in the lock domain.

5.4.4 Verification of *sparc_enable_interrupts()* function

```
 1 speconly(
 2 void VCC_SPEC_FUN_sparc_enable_interrupts(LOCK ^l claimp(c))
 3   always(c, closed(l))
 4   requires (wrapped(l→protected_obj))
 5   writes( l→protected_obj )
 6   requires( wrapped(l→protected_obj) )
 7 {
 8   atomic (c, l) {
 9     set_closed_owner(l→protected_obj, l);
10     l→interrupt_disable = 0;
11   }
12 }
13 )
```

Listing 5.15: VCC function for simulate enabling interrupts

This function is used to enable interrupts again by return queue ownership for the lock (Line 10) and update the $interrupt\_disable$ variable to 0 and enabling interrupts again.

5.4.5 Verification of *event_queue_insert_sync()* function

```
 1 int event_queue_insert_sync(struct irq_entry *irq claimp(c))
 2   always(c,closed(&QueueLock))
 3   requires(nested(&event_queue))
 4   requires(irq→data != NULL)
 5   requires(!VCC_SPEC_FUN_is_asyn_trap(irq→event))
 6   requires(thread_local(irq))
 7   reads(irq)
 8   writes(&event_queue)
 9   ensures(&event_queue.sync_event == irq)
10   ensures(nested(&event_queue))
11   ensures(result == 0 || result== 1 )
12   {
13   //xLuna assert function
14   ASSERT(irq→event);
15     assert(typed(&irq→event));
16   if (event_queue_has_sync_event()) return 1; sparc_disable_interrupts
       ();
17     spec(VCC_SPEC_FUN_sparc_disable_interrupts(spec(&QueueLock) spec(c))
         ;)
18     // unwrap/wrap protocol
19     unwrap(&event_queue);
20     unwrap(&event_queue.sync_event);
21   //insert sync event(for vcc we can now change the queue)
22   event_queue.sync_event = *irq;
23     //reverse wrapping
24     wrap(&event_queue.sync_event);
25     wrap(&event_queue);
26   sparc_enable_interrupts();
27     spec(VCC_SPEC_FUN_sparc_enable_interrupts(spec(&QueueLock) spec(c))
         ;)
28   return 0;
29 }
```

Listing 5.16: Insert synchronous event function

As preconditions we tell to VCC that this function will start in a state where:

- *event_queue* is owned by an object and thus its invariant holds (Line 3);
- *sync* events always have *data* required by event handlers to work properly (Line 4);

- event code must be within sync event bounds (Line 5);
- *irq* points to an object that is local to the running thread which is only allowed to read it. On the other hand, permissions to change *event_queue* are necessary. The *writes* clause guarantees that only *event_queue* will be updated (Lines 6, 7, 8).

To respect the VCC ownership protocol it is necessary *unwrap* (Lines 18, 19) the queue and its embedded IRQ entry, change it and then *wrap* in reverse order (Lines 23, 24). When an object is *unwrapped* VCC implicitly *assume* its invariant and *assert* it when wrap occurs. In the example, this ownership flow will guarantee the second postcondition (Line 10) whereas the first one ensures that the queue *sync* event was updated correctly (Line 9).

As we said, in VCC, knowledge can be passed to functions through ghost parameters (claims). The ghost claim parameter and the clause in Line 2 guarantee that:

- *c* has an implicit invariant ensuring that the *container* remains *closed* (*container* invariant holds)
- the *always* clause tells VCC to enforce the fact that if the *container* is *closed*, the lock is also *closed* and thus can never be destroyed.

Line 16 and 26 ensures exclusive access to the queue (see Functions 5.4.35.4.4).

5.4.6 Verification of *event_queue_insert_async()* function

```
 1 int event_queue_insert_asyn(struct irq_entry *irq claimp(c))
 2   always(c, closed(&QueueLock))
 3   requires(nested(&event_queue))
 4   requires(irq->data == NULL)
 5   requires(VCC_SPEC_FUN_is_asyn_trap(irq->event))
 6   requires(thread_local(irq))
 7   reads(span(irq))
 8   writes(&event_queue)
 9     ensures(nested(&event_queue))
10   ensures (result == 0)
11 {
12   unsigned32 *pending;
13
14   ASSERT(is_asyn_trap(irq->event));
15   assert(VCC_SPEC_FUN_is_asyn_trap(irq->event));
```

```

pending = &(event_queue.asyn_events[event_to_index(irq->event)].pending
    );

sparc_disable_interrupts(); //assembly call
spec(VCC_SPEC_FUN_sparc_disable_interrupts(spec(&QueueLock) spec(c));)
assert(me() == owner(&event_queue.asyn_events[event_to_index(irq->event
    )]));
// unwrap/wrap protocol
unwrap(&event_queue);
spec(
unsigned j;
for (j = 0; j < NUMBER_OF_ASYN_EVENTS; j++)
 invariant(j <= NUMBER_OF_ASYN_EVENTS)
 invariant(forall(unsigned i; i < NUMBER_OF_ASYN_EVENTS; i < j ?
  mutable(&event_queue.asyn_events+i) : wrapped(&event_queue.
      asyn_events+i)))
 writes(array_range(&event_queue.asyn_events,NUMBER_OF_ASYN_EVENTS))
  {
   unwrap(&event_queue.asyn_events+j);
  })
//insert asyn event
*pending = unchecked( *pending + 1);

//reverse wrapping
spec(
unsigned j;
for (j = 0; j < NUMBER_OF_ASYN_EVENTS; j++)
 invariant(j <= NUMBER_OF_ASYN_EVENTS)
 invariant(forall(unsigned i; i < NUMBER_OF_ASYN_EVENTS; i < j ?
  wrapped(&event_queue.asyn_events+i) : mutable(&event_queue.
      asyn_events+i)))
 writes(array_range(&event_queue.asyn_events,NUMBER_OF_ASYN_EVENTS))
  {
   wrap(&event_queue.asyn_events+j);
  })
wrap(&event_queue);
sparc_enable_interrupts(); //assembly call
spec(VCC_SPEC_FUN_sparc_enable_interrupts(spec(&QueueLock) spec(c));)

return 0;
```

```
53 }
```

Listing 5.17: Insert asynchronous event function

This function is similar to insert a synchronous event with the particular aspect of unwrapping the $asyn$ events array. In the same way that we needed wrap the asynchronous event array (see Function 5.6) with a ghost loop, e also need to unwrapping it using the same annotations (Line 24 to 33 and 38 to 47). Other similar annotations to the insert $sync$ function were defined. However, some aspects are different:

- Line 4. Precondition saying that $data$ is null in case of $asyn$ event.
- Line 5. The event is within bounds.
- Line 10. This function always return 0.
- Line 15. Repeat the xLuna $ASSERT$ function for VCC. Assuring that the input $irq \rightarrow event$ is an $asyn$ trap.

5.4.7 Verification of *event_queue_pop()* function

```
int event_queue_pop(struct irq_entry *irq claimp(c))
  always(c, closed(&QueueLock))
  requires(nested(&event_queue))
  requires(thread_local(irq))
  writes(irq)
  ensures(closed(&event_queue))
{
  int ret = 1;
  unsigned int i;
  sparc_disable_interrupts();

  spec(VCC_SPEC_FUN_sparc_disable_interrupts(spec(&QueueLock) spec(c));)

  // unwrap/wrap protocol
  unwrap(&event_queue);
  unwrap(&event_queue.sync_event);
  spec(
  unsigned j;
  for (j = 0; j < NUMBER_OF_ASYN_EVENTS; j++)
```

```
   invariant(j <= NUMBER_OF_ASYN_EVENTS)
   invariant(forall(unsigned i; i < NUMBER_OF_ASYN_EVENTS; i < j ?
    mutable(&event_queue.asyn_events+i) : wrapped(&event_queue.
        asyn_events+i)))
   writes(array_range(&event_queue.asyn_events,NUMBER_OF_ASYN_EVENTS))
    {
     unwrap(&event_queue.asyn_events+j);
    })//End unwrap/wrap protocol

  if (event_queue_has_sync_event()) {
    *irq = event_queue.sync_event;
    event_queue.sync_event.event = 0;
    ret = 0;
  } else {
    for (i=0; i< NUMBER_OF_ASYN_EVENTS; i++)
      {
      if (event_queue.asyn_events[i].pending) {
        event_queue.asyn_events[i].pending   ;
        irq->data = NULL;
        irq->event = i + TT_ASYN_MIN;
        assert(VCC_SPEC_FUN_is_asyn_trap(irq->event));
        ret = 0;
        break;
      }
    }
  }

  //reverse wrapping
  spec(
  unsigned j;
  for (j = 0; j < NUMBER_OF_ASYN_EVENTS; j++)
   invariant(j <= NUMBER_OF_ASYN_EVENTS)
   invariant(forall(unsigned i; i < NUMBER_OF_ASYN_EVENTS; i < j ?
    wrapped(&event_queue.asyn_events+i) : mutable(&event_queue.
        asyn_events+i)))
   writes(array_range(&event_queue.asyn_events,NUMBER_OF_ASYN_EVENTS))
    {
     wrap(&event_queue.asyn_events+j);
    })
  wrap(&event_queue.sync_event);
  wrap(&event_queue);
```

```
59
60   sparc_enable_interrupts();
61   spec(VCC_SPEC_FUN_sparc_enable_interrupts(spec(&QueueLock) spec(c));)
62
63   return ret;
64 }
```

Listing 5.18: Pop event function

The difference between the Pop function and insert sync/asyn events is that we do not know what will be event removed from the queue. Therefore, we need to $unwrap$ all the events in the $wrap$/$unwrap$ protocol (Lines 15 to 26 and 47 to 58).

- Line 5. Now we need to give write permissions to the input parameter irq instead of read permissions.
- Line 12 and 61. Guarantee exclusive access to the queue when the event is removed (disable/enable interrupts).
- Line 39. Ensures that the event removed from the queue is indeed an $asyn$ event.

The irq event removed from the queue is returned by reference in the input parameter and then sent to their handler.

All functions demonstrated until now were automatically proven to be correct using VCC tool suite against the specifications inserted in each function. The other IRQ manager function, some are related with event dispatching which its execution spans to other kernel managers outside of the scope of this thesis. Other functions manipulate the Linux stack and processor registers. To a pure verification solution for these functions, a model of the SPARC processor would be necessary. However, in Appendix A we show some annotations that were added to ensure some important functional requirements of these functions.

5.5 Summary

In this chapter we have detailed the verification work applied to xLuna kernel. All the functions presented were verified against the specifications. In the overall IRQ model (about 1000 LOC C) were inserted proximally 700 line of VCC annotations. Were verified properties such as correct array index, arithmetic overflow and pointer deference or null pointers were verified in most parts of IRQ manager. All inside function updates were surrounded by frame conditions and all parameters validated, type invariants hold with respect to the VCC ownership protocol. Our concurrency methodology ensure mutual exclusion in sensible updates abstracting from the underlying executions made by xLuna microkernel.

From this work, conclusions were pointed and the verification results gave way to future improvements to achieve a complete xLuna formal verification. The next chapter shows those results.

Chapter 6

Conclusions

In this thesis we have presented the design by contract approach to formal verification of a realistic real-time embedded operating system. This work is a collaboration effort between Critical Software and the University of Beira interior. Acquire knowledge in formal methods has been an increasing goal inside Critical Software and this work brings a useful case study applied to their own product. xLuna was design to run in highly critical environments such satellite space application and avionics. Applying formal methods to xLuna leaves the idea of a future Common Criteria security evaluation for the more demanding levels of assurance.

Design by contract approach to verify xLuna IRQ module was done in two ways using different verification tools. In [98] is shown the Frama-C approach. The results of the verification effort using VCC and Frama-C were already submitted for publication.

6.1 Results

One can instantly conclude that the annotations burden in VCC is greater than other verification tools (e.g. Frama-C), mainly due to the ghost code and type invariants supported by VCC. As said before VCC memory model guarantees that objects do not overlap in memory and played a crucial role for the safety properties verified. At this time, pre- and post-conditions were added to approximately 80% of IRQ manager C code (the remaining 20% of the code are relative to auxiliary functions outside the scope of our verification work). The functions were automatically verified against VCC specifications.

The concurrency verification approach guarantees sequential execution in a preemptable environment as required by xLuna code. Moreover, the spin lock approach was already demonstrated to be suitable to VCC methodology [15, 96] and also used in PikeOS verification[95].

6.2 VCC

VCC is being built with operating system verification in mind. However, it requires a more demanding learning curve when addressing particular low-level code and concurrency verification aspects. Nevertheless, the constant tool improvements and verification examples may lower this curve.

The VCC syntax has been changing to make it more homogeneous, simpler and more consistent. The syntax is converging to ACSL specifications (ANSI/ISO C Specification Language, already used in Frama-C).

6.3 Future Work

Future improvements for this work can be divided in three different paths.

- **Assembly code**. Assembly language in kernel code can not be ignored when aiming to an overall system verification and demanding safety standards. One possibility to extend VCC work in xLuna is the construction of a ghost underlying hardware model and assembly language to connect the specifications made to that ghost model. This would allow more precise verification of the IRQ manager and other xLuna manager.

- **SPARC** processor. A complete ghost model of the underlying architecture could enable mapping ghost assembly instructions to that model which would be also suitable for verification with VCC.

- **RTEMS**. RTEMS subsystem is the basis of the whole system. RTEMS Classic API is used in several parts of xLuna kernel and it would be desirable verify the RTEMS services used in xLuna (e.g. scheduler, thread management, etc.).

Understand unknown kernel code in a manageable time frame is not a trivial task. Therefore, to a 'pure', formally verified xLuna solution, would also be needed constant communication with xLuna development team. However, for a certified solution with the highest security standards this approach is necessary, but just one step of a complex and complete formal development cycle.

Appendices

Appendix A

Other IRQ Verified Functions

A.0.1 Verification of *irq_set_interrupt_on_off()* function

This function is used to change if Linux is to be interrupted or not. The input parameter is an an integer (0 or 1). It returns an integer with previous value of *linux_interrupt_enabled*.

```
 1 rtems_unsigned32 irq_set_interrupt_on_off(rtems_unsigned32 on_off)
 2  requires(linux_interrupt_enabled == 0|| linux_interrupt_enabled ==1)
 3  requires(on_off == 0|| on_off ==1)
 4  requires(thread_local(&linux_interrupt_enabled))
 5  writes(&linux_interrupt_enabled)
 6  ensures(linux_interrupt_enabled == on_off)
 7  ensures(result == 0 || result == 1)
 8 {
 9   rtems_unsigned32 ret = linux_interrupt_enabled;
10 
11   linux_interrupt_enabled = on_off;
12 #if 0
13   if (linux_interrupt_enabled) DEBUG_puts_no_cr("+");
14   else DEBUG_puts_no_cr(" ");
15 #endif
16   return ret;
17 }
```

Listing A.1: IRQ function to interrupt Linux

- Line 2 and 3. Preconditions state the possible values for *linux_interrupt_enabled* and *on_off* respectively.

- Line 4. Preconditions for thread locality of *linux_interrupt_enabled*.
- Line 5. Write permissions to change *linux_interrupt_enabled*.
- Line 6. Postcondition ensuring the possible results at the end of the function.

A.0.2 Verification of *irq_build_linux_regs_from_isf()* function

This function builds the Linux registers from an old CPU interrupt stack frame. One may connect possible annotation in this function to a ghost model of the underlying hardware.

```
static void irq_build_linux_regs_from_isf(struct pt_regs *regs,
    CPU_Interrupt_frame *old_isf,unsigned32 trap)
  requires(typed(regs))
  reads(old_isf)
  requires(thread_local(old_isf))
  writes(span(regs))
{
  regs -> psr = old_isf->psr;
  if (IS_KERNEL(old_isf->pc))
    regs->psr |= SPARC_PSR_PS_MASK;
  else
    ASSERT(0 == (regs->psr & SPARC_PSR_PS_MASK));
  if (is_asyn_trap(trap)) {
    regs->pc  = old_isf->pc;
    regs->npc = old_isf->npc;
  } else {
    regs->pc  = old_isf->tpc;
    regs->npc = old_isf->pc;
  }
  regs->y     = old_isf->y;
  regs->u_regs[UREG_G0] = PT_REGS_MAGIC_G0;
  regs->u_regs[UREG_G1] = old_isf->g1;
  regs->u_regs[UREG_G2] = old_isf->g2;
  regs->u_regs[UREG_G3] = old_isf->g3;
  regs->u_regs[UREG_G4] = old_isf->g4;
  regs->u_regs[UREG_G5] = old_isf->g5;
  regs->u_regs[UREG_G6] = old_isf->g6;
  regs->u_regs[UREG_G7] = old_isf->g7;
  regs->u_regs[UREG_I0] = old_isf->i0;
  regs->u_regs[UREG_I1] = old_isf->i1;
  regs->u_regs[UREG_I2] = old_isf->i2;
  regs->u_regs[UREG_I3] = old_isf->i3;
  regs->u_regs[UREG_I4] = old_isf->i4;
  regs->u_regs[UREG_I5] = old_isf->i5;
  regs->u_regs[UREG_I6] = (unsigned32)old_isf;
  regs->u_regs[UREG_I7] = old_isf->i7;
}
```

Listing A.2: IRQ function to build Linux registers from an CPU interrupt frame

- Line 2. Requires that $regs$ has to be valid in memory.
- Line 3. Read permissions to old_isf.
- Line 4. Precondition saying that old_isf points to an object that is local to this thread.
- Line 5. Write permissions to Linux $regs$.

A.0.3 Verification of *irq_build_linux_new_isf()* function

This function builds a new Linux interrupt frame. However, the stack manipulation for Linux Subsystem are out of scope. Therefore, some details of this function are assumed to be correct.

```
static void irq_build_linux_new_isf(CPU_Interrupt_frame *new_isf, struct pt_regs *regs, CPU_Interrupt_frame *old_isf, unsigned32 trap)
  requires(thread_local(new_isf))
  requires(typed(new_isf))
  writes(span(new_isf))
  requires(thread_local(old_isf))
  requires(typed(old_isf))
  reads(old_isf)
  writes(&concurrent_linux_handlers)
  requires(thread_local(&mmu_far_for_linux))
  requires(thread_local(&mmu_fstatus_for_linux))
  ensures(concurrent_linux_handlers <= MAX_CONCURRENT_LINUX_HANDLERS)
{
  memcpy(new_isf, old_isf, sizeof(CPU_Interrupt_frame));
  ASSERT(linux_interface_ptr);
  ASSERT((linux_interface_ptr[0] & 0x3) == 0);
  assume(typed(&linux_interface_ptr[0]));
assume(thread_local(&linux_interface_ptr));
assume(thread_local(&linux_interface_ptr[0]));
new_isf->pc = linux_interface_ptr[0];
new_isf->npc = unchecked(new_isf->pc+4);
new_isf->i0 = trap;
new_isf->i1 = (unsigned32)regs;
assume(typed(&linux_interface_ptr[2]));
assume(typed((unsigned int *)linux_interface_ptr[2]));
assume(thread_local((unsigned int *)linux_interface_ptr[2]));
assume(thread_local(&linux_interface_ptr[2]));
new_isf->g6 = *(unsigned int *)linux_interface_ptr[2];
if (trap == TT_MNA) {
  assume(typed((unsigned32 *)(old_isf->tpc)));
  assume(thread_local((unsigned32 *)(old_isf->tpc)));
  new_isf->i2 = *(unsigned32 *)(old_isf->tpc);
} else if (trap == TT_DATA_ACCESS) {
  unsigned int fstatus, faddr;
  faddr = mmu_far_for_linux;
  fstatus = mmu_fstatus_for_linux >> 6;
  new_isf->i2 = fstatus & 1;
  new_isf->i3 = fstatus & 2;
  new_isf->i4 = faddr & 0xfffff000;
  if (IS_RTEMS_KERNEL(old_isf->pc)) {
    ASSERT(0);
  }
} else if (trap == TT_INST_ACCESS) {
  unsigned int fstatus, faddr;
  /* The saved MMU registers for Linux */
  faddr = mmu_far_for_linux;
  fstatus = mmu_fstatus_for_linux >> 6;
  new_isf->i2 = 1;
  new_isf->i3 = 0;
  new_isf->i4 = faddr & 0xfffff000;
  if (IS_RTEMS_KERNEL(old_isf->pc)) {
                        ASSERT(0);
  }
}

new_isf->i6_fp = unchecked(((unsigned32)new_isf) + sizeof(
    CPU_Interrupt_frame));
ASSERT(concurrent_linux_handlers < MAX_CONCURRENT_LINUX_HANDLERS);
```

```
  unchecked(concurrent_linux_handlers++);
  new_isf->i7 = concurrent_linux_handlers;
}
```

Listing A.3: IRQ function to build Linux new CPU interrupt stack frame

- Line 2,3,5 an 6. Precondition saying that new_isf and old_isf point to an object that is local to this thread and that they are valid typed objects in memory.
- Line 4. Write permissions to build the new_isf.
- Line 7. Read permissions to old_isf.
- Line 8. Write permissions for $concurrent\_linux\_handlers$.
- Line 9 and 10. Precondition saying that $(mmu\_far\_for\_linux$ and $mmu\_fstatus\_for\_linux$ are local to the current thread.
- Line 11. Ensure that $concurrent\_linux\_handlers$ stays within bounds when leaving this function.
- Lines 16,17,18,23,24,25,26,29 and 30. Properties assumed to be correct.

References

[1] DO-178 Industry Group for Engineers. Website. http://www.do178site.com/, 2009. [cited at p. 1]

[2] Functional Safety and IEC 61508. Website. http://www.iec.ch/functionalsafety/. [cited at p. 1]

[3] The Common Criteria for Information Technology Security Evaluation. Website. http://www.commoncriteriaportal.org/, 2009. [cited at p. 1]

[4] B. Meyer. Applying ”design by contract”. *IEEE COMPUTER*, 25:40–51, 1992. [cited at p. 2]

[5] C. A. R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM,Pages: 576-580*, 1969. [cited at p. 2]

[6] Frama-C. Website. http://frama-c.com/. [cited at p. 2]

[7] E. Cohen, M. Dahlweid, M. Hillebrand, D. Leinenbach, M. Moskal, T. Santen, W. Schulte, and S. Tobies. VCC: A practical system for verifying concurrent C. 5674:23–42, 2009. Invited paper. [cited at p. 2, 5, 29]

[8] Gary T. Leavens and Y. Cheon. Design by contract with jml. jml tutorial. http://www.rtems.com/, 2006. [cited at p. 2, 5]

[9] K. Rustan M. Leino M. Barnett and W. Schulte. The spec# programming system: An overview. *In CASSIS 2004*, 3362, 2004. [cited at p. 2, 5]

[10] John Barnes. *High Integrity Software: The SPARK Approach to Safety and Security*. Addison Wesley, 2003. [cited at p. 2]

[11] SA. Critical Software. Website. http://www.criticalsoftware.com/. [cited at p. 2]

[12] Joaquim J. Tojal, José M. Faria, Sim ao M. de Sousa, and Carlos Carloto. Towards a formally verified kernel module. 2010. [cited at p. 3]

[13] M. Dahlweid, M. Moskal, T. Santen, S. Tobies, and W. Schulte. VCC: Contract-based modular verification of concurrent C. In *ICSE Companion 2009: 31st International Conference on Software Engineering*, pages 429–430. IEEE, May 2009. [cited at p. 6]

[14] E. Cohen, M. Moskal, W. Schulte, and S. Tobies. A precise yet efficient memory model for C. In *4th International Workshop on Systems Software Verification (SSV 2009)*, volume 254 of *Electronic Notes in Theoretical Computer Science*, pages 85–103. Elsevier Science B.V., 2009. [cited at p. 7, 28, 33]

[15] E. Cohen, M. Moskal, W. Schulte, and S. Tobies. A practical verification methodology for concurrent programs. (MSR-TR-2009-15), February 2009. [cited at p. 7, 56]

[16] E. Cohen, M. Moskal, W. Schulte, and S. Tobies. Local verification of global invariants in concurrent programs. In *Computer Aided Verification (CAV 2010)*, Lecture Notes in Computer Science, Edinburgh, UK, July 2010. Springer. To appear. [cited at p. 8]

[17] M. Zulianello P. Braga, L. Henriques. xluna:extending free/open-source real-time executive for on-bord space applications. *Small Satellites Systems and Services The ESA 4S Symposium*, 2008. [cited at p. 11]

[18] Real-Time Executive for Multiprocessor Systems (RTEMS). Website. `http://www.rtems.com/`, 2009. [cited at p. 11]

[19] Snapgear Embedded Linux. Website. `http://www.snapgear.org/`, 2009. [cited at p. 11]

[20] G. Klein. Operating system verification — an overview. *Sãdhanã*, 34(1):27–69, February 2009. [cited at p. 19, 24]

[21] SRI International. Website. `http://sri.com/`, 2009. [cited at p. 19]

[22] Peter G. Neumann and Richard J. Feiertage. Psos revisited. *ACSAC*, 2003. [cited at p. 19, 20]

[23] Peter G. Neumann and Richard J. Feiertage. The foundations of a provably secure operating system (psos). *In Proceedings of the National Computer Conference*, 1979. [cited at p. 20]

[24] P.G. Neumann, R.S Boyer, R.J. Feiertag, K.N. Levitt, and L. Robinson. A provably secure operating system: The system, its applications, and proofs. technical report csl-116. *Computer Science Laboratory, SRI International, Menlo Park, California*, 1980. [cited at p. 20]

[25] L. Robinson and O. Roubine. Special - a specification and assertion language. *Technical Report-Stanford Research Institute, Menlo Park, California*, 1977. [cited at p. 20]

[26] S. Owre, J. M. Rushby, , and N. Shankar. PVS: A prototype verification system. In Deepak Kapur, editor, *11th International Conference on Automated Deduction (CADE)*, volume 607 of *Lecture Notes in Artificial Intelligence*, pages 748–752, Saratoga, NY, jun 1992. Springer-Verlag. [cited at p. 20, 22, 25]

[27] Ford Aerospace & Communications Corporation. Kernelized secure operating system (ksos). *Executive summary for Defense Supply Service-The Pentagon*, 1979. [cited at p. 20]

[28] T. Perrine, J. Codd, and B. Hardy. An overview of the kernalized secure operating system (ksos). *In Proceedings of the Seventh DoD/NBS Computer Security Initiative Conference*, 1984. [cited at p. 20]

[29] University of California at Los Angeles. Website. `http://www.ucla.edu/`, 2009. [cited at p. 20]

[30] Bruce J. Walker, Richard A. Kemmerer, and Gerald J. Popek. Specification and verification of the ucla unix security kernel. *Commun. ACM*, 1980. [cited at p. 20]

[31] C. A. R. Hoare and Niklaus Wirth. An axiomatic definition of the programming language pascal. *IEEE Transactions On Software Engineering*, 1975. [cited at p. 20]

[32] D.I. Good, R.L. London, and W.W. Bledsoe. An interactive program verification system. *Technical Report-Stanford Research Institute, Menlo Park, California*, 1977. [cited at p. 20]

[33] William R. Bevier. A verified operating system kernel. *Report 11, Computational Logic Inc., Austin, Texas,*, 1987. [cited at p. 21]

[34] William R. Bevier. Kit: A study in operating system verification. *Technical Report 28, Computational Logic Inc., Austin, Texas,*, 1988. [cited at p. 21]

[35] R. S. Boyer and M. Kaufmann. The boyer-moore theorem prover and its interactive enhancement. *Computers & Mathematics with Applications*, 1995. [cited at p. 21]

[36] ACL2. Website. `http://userweb.cs.utexas.edu/users/moore/acl2/`, 2009. [cited at p. 21, 25]

[37] University of Technology Dresden. Website. `http://tu-dresden.de/en`, 2009. [cited at p. 21]

[38] M. Hohmuth and H. Haertig. Pragmatic nonblocking synchronization for real-time systems. *In Proceedings of the 2001 Usenix Annual Technical Conference*, 2001. [cited at p. 21]

[39] J. Liedtke. On micro-kernel construction. *In Proceedings of the 15th ACM Symposium on Operating systems principles*, 1995. [cited at p. 21]

[40] M. Hohmuth and H. Tews. The vfiasco approach for a verified operating system. *In 2nd ECOOP Workshop on Program Languages and Operating Systems*, 2005. [cited at p. 21, 22]

[41] Gerard J. Holzmann. The spin model checker: Primer and reference manual. *Addison-Wesley Professional*, 2003. [cited at p. 21]

[42] Promela Language. Website. `http://www.dai-arc.polito.it/dai-arc/manual/tools/jcat/main/node168.html`, 2009. [cited at p. 22]

[43] Endrawaty. Verification of the fiasco ipc implementation. *Thesis Report, International Master Program Computational Logic*, 2005. [cited at p. 22]

[44] H. Tews. Microhypervisor verfication: possible approaches and relevant properties. *In Proceedings of the NLUUG Voorjaarsconferentie 2007*, 2007. [cited at p. 22]

[45] E.G.H. Schierboom. Verification of fiascos ipc implementation. *Master thesis*, 2007. [cited at p. 22]

[46] P. Tullmann, J. Turner, J. McCorquodale, J. Lepreau, A. Chitturi, and G. Back. Formal methods: A practical tool for os implementors. *Proceedings of the 6th Workshop on Hot Topics in Operating Systems*, 1997. [cited at p. 22]

[47] G. Duval and J. Julliand. Modeling and verification of the rubis microkernel with spin. *In Proceedings of the First SPIN Workshop*, 1995. [cited at p. 22]

[48] T. Cattel. Modelization and verification of a multiprocessor realtime os kernel. *Proceedings of the 7th IFIP WG6.1 International Conference on Formal Description Techniques VII*, 1995. [cited at p. 22]

[49] Jonathan S. Shapiro, Jonathan M. Smith, , and David J. Farber. Eros: a fast capability system. *In In Symposium on Operating Systems Principles, pages 170-185*, 1999. [cited at p. 23]

[50] J. Shapiro, S. Sridhar, and S. Doerrie. Bitc (0.11 transitional) language specification. *BitC Documentation, the BitC Language and System*, 2008. [cited at p. 23]

[51] J. Shapiro, M. Scott Doerrie, E. Northup, and M. Miller. Towards a verified, general-purpose operating system kernel. *In 1st NICTA Workshop on Operating System Verification*, 2004. [cited at p. 23]

[52] H. Tuch, G. Klein, and G. Heiser. Os verification — now! In Margo Seltzer, editor, *Proc. 10th Workshop on Hot Topics in Operating Systems (HotOS X)*, 2005. to appear. [cited at p. 23]

[53] J. Liedtke, U. Dannowski, K. Elphinstone, G. Liefländer, E. Skoglund, V. Uhlig, C. Ceelen, A. Haeberlen, and M. Völp. The l4ka vision, April 2001. [cited at p. 23]

[54] NICTA-Australia's ICT Research Centre of Excellence. Arm architecture and systems. `http://www.cse.unsw.edu.au/~disy/L4/MIPS/`, 2001. [cited at p. 23]

[55] NICTA-Australia's ICT Research Centre of Excellence. Secure microkernel project (sel4). `http://ertos.nicta.com.au/research/sel4/`, 2009. [cited at p. 23]

[56] Leonid Ryzhyk. The arm architecture. 2006. [cited at p. 23]

[57] Haskell programming language. Website. `http://www.haskell.org/`, 2009. [cited at p. 23]

[58] Isabelle proof assistant. Website. `http://isabelle.in.tum.de`, 2009. [cited at p. 24]

[59] G. Klein, K. Elphinstone, G. Heiser, J. Andronick, D. Cock, P. Derrin, D. Elkaduwe, K. Engelhardt, R. Kolanski, M. Norrish, T. Sewell, H. Tuch, and S. Winwood. seL4: Formal verification of an OS kernel. In *Proc. 22nd ACM Symposium on Operating*

Systems Principles (SOSP), pages 207–220, Big Sky, MT, USA, October 2009. ACM. [cited at p. 24]

[60] G. Klein. Correct OS kernel? proof? done! *USENIX ;login:*, 34(6):28–34, Dec 2009. [cited at p. 24]

[61] X. Feng, Z. Shao, Y. Guo, and Y. Dong. Certifying low-level programs with hardware interrupts and preemptive threads. *Journal of Automated Reasoning (Special Issue on Operating System Verification) 42 (2-4): 301-347*, 2009. [cited at p. 24]

[62] specware tool. Website. `http://www.specware.org/`, 2007. [cited at p. 24]

[63] W. Martin, P. White, F. S. Taylor, and A. Goldberg. Formal construction of the mathematically analyzed separation kernel. *Proceedings of the 15th IEEE international conference on Automated software engineering*, 2000. [cited at p. 24]

[64] D. Greve and M. Wilding. A separation kernel formal security policy. *Rockwell Collins, Inc. Advanced Technology Center*, 2000. [cited at p. 25]

[65] Information Assurance Directorate. U.s. government protection profile for separation kernels in environments requiring high robustness. *IAD,Version 1.03*, 2007. [cited at p. 25]

[66] David Greve, Raymond Richards, and Matthew Wilding. A summary of intrinsic partitioning verification. *In In Proceedings of the Fifth International Workshop on the ACL2 Theorem Prover and Its Applications*, 2004. [cited at p. 25]

[67] C. Heitmeyer, M. Archer, E. Leonard, and J. McLean. Formal specification and verification of data separation in a separation kernel for an embedded system. *Proceedings of the 13th ACM conference on Computer and communications security,Pages: 346-355*, 2006. [cited at p. 25]

[68] C. Heitmeyer, M. Archer, E. Leonard, and J. McLean. Applying formal methods to a certifiably secure software system. *IEEE Transactions on Software Engineering,Pages: 82-98*, 2008. [cited at p. 25]

[69] United States Naval Research Laboratory. Website. `http://www.nrl.navy.mil/`, 2009. [cited at p. 25]

[70] M. Archer, C. Heitmeyer, and E. Riccobene. Using tame to prove invariants of automata models: Two case studies. *Proceedings of the third workshop on Formal methods in software practice,Pages: 25 - 36*, 2000. [cited at p. 25]

[71] M. Archer, E. Leonard, and M. Pradella. Analyzing security-enhanced linux policy specifications. *Proceedings of the 4th IEEE International Workshop on Policies for Distributed Systems and Networks*, 2003. [cited at p. 25]

[72] G. Hunt, J. Larus, M. Abadi, M. Aiken, P. Barham, M. Fhndrich, C. Hawblitzel, O. Hodson, S. Levi, N. Murphy, B. Steensgaard, D. Tarditi, T. Wobber, and B. Zill. An overview of the singularity project. *Microsoft Research Technical Report MSR-TR-2005-135*, 2005. [cited at p. 26]

[73] Galen C. Hunt and James R. Larus. Singularity: rethinking the software stack. *ACM SIGOPS Operating Systems Review,Pages: 37 - 49*, 2007. [cited at p. 26]

[74] M. Fhndrich, M. Aiken, C. Hawblitzel, O. Hodson, G. Hunt, J. Larus, and S. Levi. Language support for fast and reliable message-based communication in singularity os. *ACM SIGOPS Operating Systems Review,Pages: 177 - 190*, 2006. [cited at p. 26]

[75] Spec# Microsoft Research. Website. `http://research.microsoft.com/en-us/projects/specsharp/`, 2009. [cited at p. 26]

[76] M. Barnett, B. Evan Chang, R. Deline, B. Jacobs, and K. Rustanm. Leino. Boogie: A modular reusable verifier for object-oriented programs. In *Formal Methods for Components and Objects: 4th International Symposium, FMCO 2005, volume 4111 of Lecture Notes in Computer Science*, pages 364–387. Springer, 2006. [cited at p. 26]

[77] Green Hills Integrity real-time Operating System. Website. `http://www.ghs.com/products/safety_critical/integrity-do-178b.html`, 2009. [cited at p. 26]

[78] Wind River's VxWorks real-time Operating System. Website. `http://www.windriver.com/products/vxworks/index.html`, 2009. [cited at p. 26]

[79] The Verisoft Consortium. Website. `http://www.verisoft.de`, 2008. [cited at p. 27]

[80] The Verisoft XT Consortium. Website. `http://www.verisoftxt.de/`, 2009. [cited at p. 27]

[81] S. Beyer, C. Jacobi, D. Kroening, D. Leinenbach, and W. Paul. Putting it all together: Formal verification of the VAMP. *International Journal on Software Tools for Technology Transfer*, 8(4–5):411–430, August 2006. [cited at p. 27]

[82] M. Hillebrand and S. Tverdyshev. Formal verification of gate-level computer systems. In Anna Frid, Andrey Morozov, Andrey Rybalchenko, and Klaus W. Wagner, editors, *Computer Science – Theory and Applications, Fourth International Computer Science Symposium in Russia, CSR 2009, Novosibirsk, Russia, August 18–23, 2009, Proceedings*, volume 5675 of *Lecture Notes in Computer Science*, pages 322–333. Springer, 2009. [cited at p. 27]

[83] T. In der Rieden and A. Tsyban. CVM – A verified framework for microkernel programmers. In *3rd International Workshop on Systems Software Verification (SSV 2008)*, volume 217C of *Electronic Notes in Theoretical Computer Science*, pages 151–168. Elsevier Science B.V., 2008. [cited at p. 27]

[84] M. Daum, J. Dörrenbächer, and S. Bogan. Model stack for the pervasive verification of a microkernel-based operating system. In Bernhard Beckert and Gerwin Klein, editors, *Proceedings, 5th International Verification Workshop (VERIFY), Sydney, Australia*, volume 372 of *CEUR Workshop Proceedings*, pages 56–70. CEUR-WS.org, August 2008. [cited at p. 27]

[85] S. Bogan. *Formal Specification of a Simple Operating System.* PhD thesis, Saarland University, Computer Science Department, August 2008. [cited at p. 27]

[86] G. Beuster, N. Henrich, and M. Wagner. Real world verification – Experiences from the Verisoft email client. In Geoff Sutcliffe, Renate Schmidt, and Stephan Schulz, editors, *Proceedings of the FLoC'06 Workshop on Empirically Successful Computerized Reasoning (ESCoR 2006)*, volume 192 of *CEUR Workshop Proceedings*, pages 112–125. CEUR-WS.org, August 2006. [cited at p. 27]

[87] Inc. Advanced Micro Devices (AMD). Amd64 architecture programmers manual:volumes 13. 2006. [cited at p. 28]

[88] Intel Corporation. Intel 64 and ia-32 architectures software developers manual:volumes 13b. 2006. [cited at p. 28]

[89] D. Leinenbach and T. Santen. Verifying the Microsoft Hyper-V Hypervisor with VCC. In Jean-Raymond Abrial, Michael Butler, Rajeev Joshi, Elena Troubitsyna, and Jim

C. P. Woodcock, editors, *09381 Extended Abstracts Collection – Refinement Based Methods for the Construction of Dependable Systems*, number 09381 in Dagstuhl Seminar Proceedings, pages 104–108, Dagstuhl, Germany, 2010. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Germany. [cited at p. 28, 29]

[90] S. Maus, M. Moskal, and W. Schulte. Vx86: x86 assembler simulated in C powered by automated theorem proving. In José Meseguer and Grigore Roşu, editors, *Algebraic Methodology and Software Technology (AMAST 2008)*, volume 5140 of *Lecture Notes in Computer Science*, pages 284–298, Urbana, IL, USA, July 2008. Springer. [cited at p. 28, 43]

[91] SYSGO embedding innovations. Website. `http://www.sysgo.com/`, 2007. [cited at p. 29]

[92] C. Baumann and T. Bormer. Verifying the PikeOS microkernel: First results in the Verisoft XT Avionics project. In Ralf Huuck, Gerwin Klein, and Bastian Schlich, editors, *Doctoral Symposium on Systems Software Verification (DS SSV 2009)*, number AIB-2009-14 in Aachener Informatik Berichte, pages 20–22. Department of Computer Science, RWTH Aachen, June 2009. [cited at p. 29]

[93] C. Baumann, B. Beckert, H. Blasum, and T. Bormer. Formal verification of a microkernel used in dependable software systems. In Bettina Buth, Gerd Rabe, and Till Seyfarth, editors, *Computer Safety, Reliability, and Security (SAFECOMP 2009)*, volume 5775 of *Lecture Notes in Computer Science*, pages 187–200, Hamburg, Germany, 2009. Springer. [cited at p. 29]

[94] C. Baumann, B. Beckert, H. Blasum, and T. Bormer. Better avionics software reliability by code verification – A glance at code verification methodology in the Verisoft XT project. In *Embedded World 2009 Conference*, Nuremberg, Germany, March 2009. Franzis Verlag. To appear. [cited at p. 30]

[95] C. Baumann, B. Beckert, H. Blasum, and T. Bormer. Ingredients of operating system correctness. In *Embedded World 2010 Conference*, Nuremberg, Germany, March 2010. To appear. [cited at p. 30, 56]

[96] Mark A. Hillebrand and Dirk C. Leinenbach. Formal verification of a reader-writer lock implementation in C. In *4th International Workshop on Systems Software*

Verification (SSV 2009), volume 254 of *Electronic Notes in Theoretical Computer Science*, pages 123–141. Elsevier Science B.V., 2009. [cited at p. 30, 56]

[97] Inc. SPARC International. The sparc architecture manual. [cited at p. 46]

[98] C. Carloto. Towards a formally verified kernel using the frama-c. *Master thesis*, 2010. [cited at p. 55]