UNIVERSIDADE DA BEIRA INTERIOR
Covilhã | Portugal

Towards a Formally Verified Microkernel using the Frama-C toolset

Carlos José Abreu Dias da Silva Carloto

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

Supervised by Simão Melo de Sousa

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

Acknowledgements

First of all I want to thanks to everyone that ever supported me since the beginning of my studies until now.

This part of the dissertation is by far the most complicated. There are so many people to thank that it is impossible to name all.

So, starting with the most obvious, I want to thank to my parents, António José and Maria do Céu, that have always support me. They have always give me the best conditions possible so that I can achieve my goals, and for that a big Thank You to them.

Next I will thank to the rest of my family. They were always there when I needed the most. In the family subject, I want to give a special thanks to Carlota. She is my godson and had always bring entertainment to my life.

Now, I want to thank to my colleges from the Master of Science (MSc) course and specially to Joaquim Tojal. He was my partner in this dissertation and helped me a lot. A special thanks to my supervisor, professor Simão Melo de Sousa that always give me the right direction to go in the investigation.

I also want to thanks to Critical Software, SA (CSW) for the support given.

To finish, the last but not least, I want to give a big, special Thank You to my girlfriend Andreia. She was the best support that I could have. I also want to wish her the best of lucks in her studies.

Abstract

This dissertation is included in the MSc course in Computer Science of the University of Beira Interior. It is a Formal Method's related dissertation, where it's used an Hoare Logic based paradigm, the Design by Contract (DbC).

This project consists in doing a Formal Verification of an industrial real-time Operating System (OS) kernel. The OS kernel that is verified is the eXtending free/open-source reaL-time execUtive for oN-board space Applications (xLuna). It is an OS from a portuguese company, CSW. The code that was verified is the real source code of xLuna. More precisely the source code of the Interrupt request (IRQ) Manager module.

The platform that was used to do the verification is the FRAmework for Modular Analyses of C (Frama-C) Toolset which is a platform that allows the verification of C code. Some incompatibilities were found in the use of the Frama-C in the source code of the IRQ Manager. Both results and Frama-C incompatibilities will be analyzed in the dissertation.

Keywords

Design By Contract, Formal Verification, xLuna, Formal Methods, Frama-C, Hoare Logic, Static Verification, Deductive Verification, Separation Kernel

Contents

List of Figures

List of Tables

Acronyms

xLuna eXtending free/open-source reaL-time execUtive for oN-board space Applications

Frama-C FRAmework for Modular Analyses of C

IRQ Interrupt request

RTEMS Real-Time Executive for Multiprocessor Systems

MSc Master of Science

OS Operating System

ACSL ANSI/ISO C Specification Language

ESA European Space Agency

ISC Inter-System Communication

VCC A Verifier for Concurrent C

JML Java Modeling Language

API Application programming interface

DbC Design by Contract

INRIA Institut National de Recherche en Informatique et en Automatique

CEA Commissariat a la Energie Atomique

CSW Critical Software, SA

UCLA University of California at Los Angeles

PVS	Prototype Verification System
EAL	Evaluation Assurance Level
NRT	Non Real-Time
HRT	High Real-Time
SOS	Simple Operating System
VFiasco	Verified Fiasco
KIT	Kernel for Isolated Tasks
DAL	Design Assurance Levels

Chapter 1

Introduction

This is a project inserted in the MSc course and it is the dissertation that concludes it. This MSc course was orientated to the formal methods, artificial intelligence and informations security. In continuation to the work done in the project of the under-graduate course, that was a project based in the formal methods [1]. This dissertation is also related to the formal methods. In particularly it is related to the DbC [1, 2, 3]. It is a formal software methodology, based on the Hoare logic paradigm [4, 5, 6].

In the beginning, we have the idea of making a formal verification of a critical system. After making an analyze of the critical system to verify, we made contact with some projects that has been done in verification of OS Kernel's. Almost every computer system depends directly on OS behavior. So, the verification of an OS Kernel is an almost perfect and challenging target to a Formal Verification.

An OS Kernel is a critical application, and on these applications it is hard to know how much trust we can put on the application. So, the common criteria [7] for information technology security evaluation defines various levels of assurance. These levels go from Evaluation Assurance Level (EAL)-1 to EAL-7. Now a little explanation of those levels:

- *EAL-1*: Functionally Tested.
- *EAL-2*: Structurally Tested.
- *EAL-3*: Methodically Tested and Checked.
- *EAL-4*: Methodically Designed, Tested and Reviewed.

[1]This project was about the formal development of a voting system using the DbC paradigm

- *EAL-5*: Semi formally Designed and Tested.
- *EAL-6*: Semi formally Verified, Designed and Tested.
- *EAL-7*: Formally Verified, Designed and Tested.

Even the highest assurance level, EAL-7, does not require formal verification of the system implementation. However there is the tag *fully formally verified* that guarantees that the system is 100% functionally correct. That means that the implementation always strictly follows high level abstract specification of kernel behavior. This guarantee that the kernel never crashes and never performs unsafe operations. But we can say even more: the behavior of the kernel in every situation is precisely predictable.

However in the avionic industry there is another security evaluation, the D0-178B, that is the safety and reliable standard in avionics and aerospace systems. This standards provides different safety levels, Design Assurance Levels (DAL). They are defined on the basis of potential effects on flight safety, having the following degrees of criticality: catastrophic, dangerous, major, minor, and no effect. Catastrophic criticality corresponding to DAL A is the highest and no effect criticality corresponding to DAL E is the lowest. In DAL the utilization of the Formal Methods are only highly recommended actually. In a near future it is very probable that the use of Formal Methods will be required.

Now that we have a target to verify we needed to choose the platform that will be used in the verification. There were various platforms that implement the DbC paradigm. Now I will list some of them:

- Spec# [8].
- Java Modeling Language (JML) [9].
- A Verifier for Concurrent C (VCC) [10].
- Eiffel [11].
- Ruby-contract [12].
- Frama-C [13].

The first idea was to use the Frama-C platform, previously Caduceus, which is a static and deductive verifying platform, and it is based in the DbC paradigm. After analyzing Frama-C, we thought that it was the perfect platform to my project. The Frama-C platform is co-developed by two French public institutions:

- Commissariat a la Energie Atomique (CEA)-LIST and
- Institut National de Recherche en Informatique et en Automatique (INRIA)-Saclay.

The next task was to choose what OS Kernel will be verified. At this moment a portuguese company, CSW, launched the challenge to do the verification of their OS, the xLuna. After that, we had a target to verify and a platform to do the verification. After this, we started to study the xLuna and decided to do the verification in small steps, module by module.

So the chosen module to start the verification was the xLuna IRQ Manager. With the verification of this module we expect to learn the lessons that will make the verification of the other modules easier.

So, the work made on this dissertation was the static and deductive verification of the IRQ Manager, which is a module of xLuna OS. To do this verification we used the Frama-C platform.

Now i will made a small summary of the organization of the dissertation:

- *Chapter 2*: The description of the tools used in the verification.
- *Chapter 3*: The State of the Art of OS verification.
- *Chapter 4*: The introduction and analyze of xLuna.
- *Chapter 5*: The verification chapter with the presentation and analyze of the contracts.
- *Chapter 6*: Conclusions of the verification and of the use of Frama-C and Future Work.

Chapter 2

Tools

In this chapter the description of the tools used in the project will be made. The tools used were:

- *Frama-C* and its plug-ins:

  1. *Jessie*
  2. *Value analysis*
  3. *Occurrence analysis*
  4. *Sparecode Analysis*

- *Why Platform*

- *Alt-Ergo Prover*

2.1 Static Analysis

Static analysis [14, 15] is the inference of behavioral essence of a program without in fact requiring its execution. It consists in particular inspection of the code. In the context of program verification it is used with the objective of finding possible errors or detecting possible anomalies in the code or simply inspecting its behavior. Several of such analyses are available in the Frama-c toolset, but also in the following:

- BLAST
- Clang

- Sparse
- Splint

Some of the techniques used in the static analysis are: model checking, data-flow analysis, symbolic analysis and abstract interpretation.

Abstract interpretation is heavily use in the Frama-C framework. It consists in a abstract machine that approximates the behavior of the program in analyze.

2.2 Frama-C

2.2.1 Frama-C introduction

Frama-C [13] is a platform that provides a collaborative environment for several C program analyses (that come in the form of plug-ins). These analyses are of two type:

1. Static Analysis: as previously introduced, these analyses compute certain aspect of the behavior of the inspected program that are useful for the verification.

2. Deductive Analysis: these analysis allow a hoare-logic based deductive verification of the inspected code. The main deductive plug-in is the Jessie plug-in that will be detailed in the next section.

The annotations in Frama-C are written in ACSL. For this work, we make heavy use of the deductive verification plug-in. Figure 2.1 depicts how Frama-C platform works.

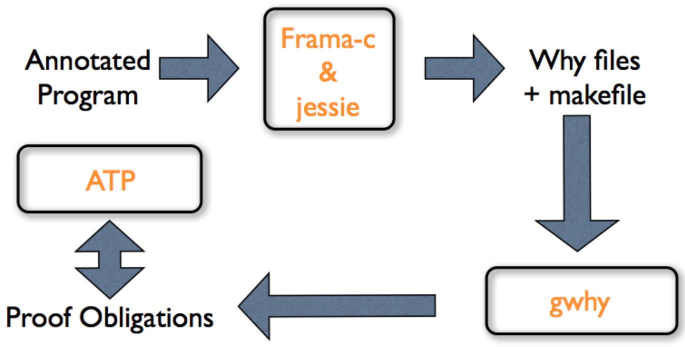


Figure 2.1: Frama-C platform. Excerpt from http://frama-c.com/

2.2.2 Deductive Analysis in Frama-C

The deductive verification is done through Jessie [16], which is a plug-in of Frama-C. It is the Frama-C plugin that enables design by contract development of C programs. The contracts are written in the ACSL. The generated verification conditions can be submitted to external automatic provers such as Simplify [17], Alt-Ergo [18], Z3 [19], Yices [20], and CVC3 [21]. So, to summarize, the Jessie plug-in takes a C annotated program and convert it in a file that can be used in the Why platform.

2.2.3 Static Analysis in Frama-C

Among the several available static analysis plug ins, we highlight those that we use in our verification task:

- *Value analysis*
- *Occurrence analysis*
- *Sparecode Analysis*

The value analysis plug-in control the values of all the variables in the program. It tries to find variables that may not have the right behavior.

In the Occurrence analysis we can track a variable along the program. We pick an variable and the tool searches all the code and shows where that variable is used.

The spare code plug-in simply analysis the code looking for dead code.

2.2.4 ACSL

The ACSL [22] is the specification language used in Frama-C. With ACSL we can add pre and post conditions and invariants. This contracts are written in C comments. So we can compile the annotated program with any C Compiler. Example:

```
/*@ requires \valid(p);
 @ assigns *p;
 @ ensures *p == \old(*p) + 1;
 @*/
void incrstar ( int *p);
```

Listing 2.1: ACSL Example

The contract is given by the comment, which starts with /*@. I ends with @*/. In the example we have a precondition in the first line. In the second line an assigns clause, that indicates that the pointer p is in a safely memory location. The third line is an poscondition that ensures that the pointer p will have its old value incremented by one.

2.3 Why platform

The Why tool [23] takes annotated programs written in a very simple imperative programming language of its own, produces verification conditions and sends them to existing provers (proof assistants such as Coq [24], PVS [25] etc. or automatic provers such as Simplify, Alt- Ergo, etc.). In this project it was used the Alt-Ergo. In the figure 2.2 it is possible to see how the why platform works on Frama-C.

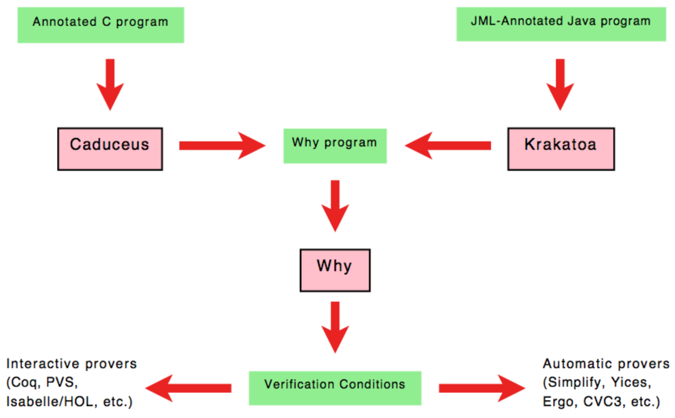

Annotated C program
JML-Annotated Java program
Caduceus
Why program
Krakatoa
Why
Interactive provers (Coq, PVS, Isabelle/HOL, etc.)
Verification Conditions
Automatic provers (Simplify, Yices, Ergo, CVC3, etc.)

Figure 2.2: Why platform. Excerpt from: http://ralyx.inria.fr/

2.4 Alt-Ergo prover

It is an automatic theorem prover used in program verification. Alt-ergo is a SMT prover and contains a SAT-solver and an instantiation mechanism. In the figure 2.3 we can see Alt-ergo overall architecture.

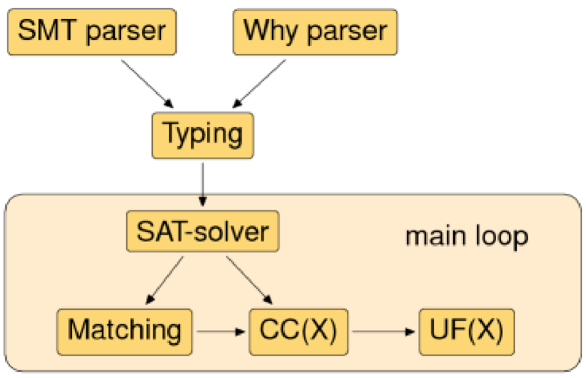


Figure 2.3: Alt-ergo Architecture. Excerpt from ergo.lri.fr/

2.5 Summary

This was the description of the tools used in the verification of the IRQ Manager that will be analyzed in the chapter 5.

Chapter 3

Related Work

In this chapter, the State of the Art of OS Verification will be presented. To start it will be shown a table that have the five most important verifications done on an OS. Next the work that was done in those projects will be explained.

So, this chapter will have the following structure:

- An overview of the OS verification projects.
- The explanation of the following projects:
    1. UCLA Secure Unix.
    2. KIT.
    3. VFiasco/Rodin.
    4. Verisoft.
    5. L4 Verified/seL4.

The work done in this section is highly based in the work of Gerwin Klein [26, 27, 28, 29] that, among an impressive work on OS Verification, provides an excellent state of the art and survey of related work.

3.1 Overview

The projects that will be presented in this chapter, are only the projects that aim to prove functional correctness or security requirements of an OS implementation. The table 3.1 [29] that is presented are organized as follows:

UCLA Secure Unix	Security Model	90%	XIVUS	1980
KIT	Isolated Tasks	100%	Boyer Moore	1987
VFiasco/Rodin	Does Not Crash	70%	Prototype Verification System (PVS)	2001 - 2008
Verisoft	Application Level	100%	Isabelle	2004 - 2008
L4 Verified	Security Model	100%	Isabelle	2005 - 2008

Table 3.1: OS Verification Overview

- The first column has the name of the project.
- The second column contains the level of the verification.
- The third column contains the amount of specification that was made.
- The fourth column indicates the prover used.
- Finally, the fifth column contains the year of the project.

Now, in the next sections those five projects will be analyzed.

3.2 UCLA Secure Unix

UCLA Secure Data Unix [30, 31, 32] is an OS that aimed to provide a standard UNIX interface to applications. The verification concentrate on the OS Kernel that is very similar to the modern microkernels. This OS can provide:

- Threads.
- Access Control.
- Virtual Memory.
- Input/Output Devices.

The proof technique used was the formal refinement. The idea it is to increase the detail of a more abstract layer that as the same behavior of the new more concrete layer. This technique is an instance of the forward simulation, that is a more general technique. As an example of this technique we can say *pick a object*, and on a more concrete layer *pick a red object*. In this example the refinement step used was reducing the non-determinism.

As was on the table only about 90% of the specifications were completed. This happen because the XIVUS [33] did not support pointers, so the functions that use pointers were not covered by the simplified version of Pascal [34] that was implemented. They had isolated all the functions that required access to pointers. However this functions were annotated with pre and postconditions. To this verification one of the most important component was the invariants, that needed to be true before and after every kernel call, but not during the call.

In this project, the authors refer that the built of the contracts were painful and tedious [30]. They also refer that the effort that they had on the specification and verification process was less than the time spent on design, implementation and debugging.

3.3 KIT

KIT [35, 36] is a short OS kernel with a simple von Neumann architecture [37]. KIT provides:

- Access to asynchronous Input/Output Devices.
- Exception handling.
- Single-word message passing.

But it fails to provide shared memory, dynamic creation of processes and file systems.

KIT's implementation has 620 lines of assembler source code and 300 lines of assembler instructions. It is a lot smaller than the modern microkernels.

It was a really important OS because it was the first ever kernel that was formally verified.

For this verification it was used the ACL2 [38] prover predecessor, the Boyer-Moore theorem prover [39, 40]. This prover was criticized because it was really hard to read. That happens because this theorem is very similar to pure LISP. And the parenthesis prefix form used on LISP language was what caused the specification so hard to read.

The proof of KIT was very similar to the proof of the UCLA Secure Unix.

After this two verifications, a long time passed into the next real attempt of verifying the implementation of an OS. This verifications were really important because they demonstrate that it was possible to verify an OS Kernel

3.4 VFiasco

The start of this project was in 2001, 14 years after the last project about OS verification. Fiasco [41] is a re-implementation of the L4 microkernel. In this implementation it was improved the following properties:

- Reliability.
- Maintainability.
- Real-Time Properties.

A small part of the L4 [42] Kernel was also removed. Fiasco implementation language was c++ and an optimized IPC was used in assembly. The following issues were related to be fundamental for the verification of modern microkernels:

1. As accurate formal semantics of the implementation language.
2. The view of the memory used in kernel code is more complex than the memory used in other applications.

The solution to the second point was to create an invariant to control the behave of the memory. But it can be broken temporarily, and on that case a more complex semantics are needed [43, 44] .

The theorem prover used in this project was the PVS [25], and for that it was necessary to translate the program directly to its semantics.

In the final of this project it was produced something about 70% of the specification. The implementation was barely verified.

3.5 Verisoft

This project started in 2003 and was funded by the German Federal Ministry of Education and Research. The initial project had the duration of 4 years. In 2007, when the project ended, immediately after, the successor of this project began. Its name was Verisoft XT. The objective of this project is to reach a pervasive formal verification starting on the application layer into the hardware, including a microkernel and a compiler. These layers produce various abstraction levels. More details and invariants are necessary as the level of abstraction is lower, in order to ensure overall system correctness.

The part of the OS verification of the Verisoft Project [45, 46] is estimated to last at least 30 years.

Now, I will explain the verification approach. It is a layered approach that goes from the hardware into the software. The verification is implemented and formally specified. A great number of them are verified in pen and paper proofs. The proofs checked on the Isabelle/HOL prover [47] were about the 75% of the proofs. Now, the description of the layers from the bottom to the top:

- *Hardware*: It has a VAMP microprocessor [48], that is a real processor, but it is not widely used. It was formally verified in the PVS theorem prover [25] before the Verisoft project started.

- *Communicating Virtual Machines*: This layer sets the hardware independent interface for the rest of the kernel to run on. Because of this division the verification effort is simpler. It will isolate the assembly level constructs.

- *VAMOS* [28]: It is the code that runs in the privilege mode of the hardware. Joining this layer with the previous we have the OS Kernel. It also include a kernel mode device driver and a memory paging. The verification of this layer is not as advanced as the verification of the Communicating Virtual Machines layer.

- *Simple Operating System (SOS)* [49, 50, 51]: It is implemented in the user mode of the hardware and runs in privileged user process. This OS have:

  1. File based input/output.
  2. IPS.
  3. Sockets.
  4. Remote procedure calls to the application level.
  5. *Only part of the SOS has been verified..*

- *Application Layer*: It only has example applications, some of them already formally verified.

The technology used to the main code verification, was a generic environment in the theorem prover Isabelle [47]. That tool includes a Floyd Hoare style logic.

The focus of the Verisoft was on pure implementation correctness. It does not do any investigations in high level security policies or access control models of the OS.

3.6 L4 Verified/seL4

In this verification we had two different projects that started in 2004:

1. *L4 Verified*: Provide a machine-checked.

2. *seL4* [52]: Formal Correctness proof of the seL4 microkernel [26].

3.6.1 seL4

By having this two parallel projects they had the goal of achieving an implementation correctness proof for seL4 [26]. The seL4 project was completed with success in the end of 2007. It provides the following services:

- Threads.

- IPC.

- Virtual Memory.

- Capabilities.

- Interrupt Control.

It was implemented in C and assembly. The performance of the seL4 was in some cases even better than the performance of the L4, that is currently the fastest microkernel. The capabilities of the seL4 are software implemented on a standard processor architecture.

The great integration of the teams in the two projects was a real important factor in the success of design of the seL4 kernel [26], especially because the early feedback from the team that was verifying L4 was very useful to the design team.

The prototype seL4 was written in Haskell [53], a pure functional language that is very similar to the notation of the theorem prover, so it can be automatically translated into the theorem prover. All the work of designing, implementing and validating take 6 years to complete.

3.6.2 L4 Verified

The refinement approach to the L4 Verified [54, 55] was similar to the refinement approach of the UCLA Secure Unix. In the bottom it has the High Performance C Implementation. Above it has the low level design. Next it has the high level design. And finally, at the top, the access control model. All these specifications are completed. In the total, it has around 20800 lines of code in total. 10000 lines of code are from the C implementation. In this implementation it was used the true subset of C, so in this way, it can me compiled with standard tools. However the following features of C were restricted:

- Taking address of a local variable is disallowed.
- Unions are not allowed.
- Functions Pointers are also not allowed.

In the global of the project including tool, library, logic and C model development, it took an estimated 10 years to complete.

3.7 Summary

To conclude this chapter, there is only one OS Kernel that have received the tag *fully formally verified*. That was the seL4 [26]. There are few systems that have the EAL7 certification.

Being xLuna an OS that is used in the space industry, the achievement of a DAL-A is clearly a commercial and important goal of CSW.

Looking to the projects described above, our project is a little bit different because we made a static and deductive approach. Also, the target of this dissertation is not all the xLuna kernel but only a module. In this dissertation all the proofs were verified automatically. In the projects described previously there are also a manual verification of some proofs.

Chapter 4

xLuna

In this chapter the description of xLuna will be made. xLuna is a separation kernel. A little introduction to separation kernel is made before the xLuna description. Then, it will be made the global description of xLuna and also a more detailed description of the target of our verification, the IRQ Manager.

4.1 Separation Kernel

4.1.1 Overview

A separation kernel [56] is nothing more than a minimal OS kernel that does just the necessary to create separate execution partitions. It also controls the communications between the partitions. A separation kernel normally manages:

1. *Hardware protection mechanisms*
2. *Fields interrupts*
3. *Performs scheduling*

All applications are made within and across partitions. This allows the application security to be separated in two parts:

- Separated and controlled communications created by the separation kernel.
- How the resources are used by any application.

The minimality of this approach permits construction of a credible assurance argument.

4.1.2 Partitioning Kernel

The partitioning kernel [57] is the part of the OS that enforces data isolation and controls the information that flow in the memory partitions. In the memory partitions (where software programs and data are), only authorize information flows between the partitions. Some OS functions will be outside the partition kernel. The partitioning kernel is used in embedded real-time systems. It is used in those systems because those systems have strict safety and security requirements.

Some of the characteristics expected to appear in the partition kernel are:

- Real-time.
- Deterministic.
- Multiple partitions.
- Hardware-supported data isolation techniques.

4.2 xLuna

4.2.1 Overview

xLuna [58] is an OS created by CSW to provide European Space Agency (ESA) an open source real-time kernel. The solution to the creation of this OS was to integrate the real-time OS Real-Time Executive for Multiprocessor Systems (RTEMS) [59] and the most used open source kernel, the Linux. The Linux kernel runs on unprivileged mode.

To summarize, xLuna kernel has two main sub-systems:

- RTEMS: runs High Real-Time (HRT) tasks in privileged-mode.
- Linux kernel: that runs Non Real-Time (NRT) tasks executed on a restricted mode.

By having this separation, xLuna can activate or de-activate the Linux sub-system at any time. It can also protect xLuna from erroneous behavior of the Linux kernel.

The RTEMS kernel is a trustable OS that has the following advantages:

1. Maintainability
2. Robustness
3. Standard Compliance
4. Performance

All this advantages makes RTEMS a great choice to use in space missions.

xLuna was designed to supports ESA's LEON SPARC processor 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. Its general architecture is shown in Figure 4.1.

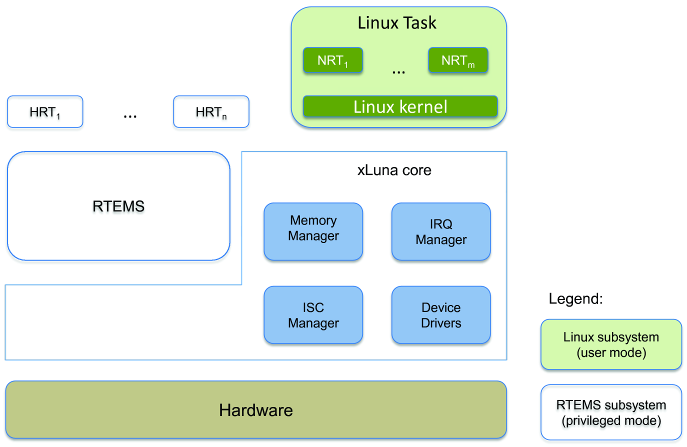


Figure 4.1: xLuna architecture

xLuna provides four main modules [58]:

- *Memory Manager*: enforces the isolation requirements between RTEMS and Linux and memory protection of Linux kernel from NRT user processes.
- *IRQ Manager*: connects interrupts of the real hardware to the Linux kernel (which does not have access to them) and provides services.

- *Inter-System Communication (ISC) Manager*: for bidirectional communication between HRT and NRT tasks.
- *Device Drivers*: for other hardware virtualizations required (e.g. timer).

The IRQ Manager was the target of the verification.

4.2.2 IRQ Manager

The principal objective of the IRQ Manager is to connect interrupts of the hardware into the Linux kernel. At the same time, it has to ensure that the amount of time that HRT Task-dispatching is disabled is the minimum possible.

Those are the main features provided by the IRQ Manager are:

- Incoming hardware interrupts must be managed by the Linux kernel, by catching these interrupts and pipelining the corresponding RTEMS handlers.
- Redirect all Linux traps to the correct handlers.
- It allows Linux kernel to virtually disable interrupts.
- Take care of the time synchronization for the Linux subsystem.
- Providing RTEMS API for managing the Linux subsystems.

So, the IRQ Manager have the following four components:

1. *dispatcher*
2. *IRQ monitor task*
3. *IRQ system call services*
4. *IRQ manager API*

4.2.2.1 Dispatcher

It is the entry point of the interrupts. It has to register the events in the event queue for the interrupts that are needed for the Linux subsystem to work well. The dispatcher is the most fundamental part of xLuna kernel because it has to minimize the latencies of the HRT Tasks.

4.2.2.2 IRQ monitor task

All the events processed by the dispatcher are processed by this monitor. The IRQ monitor task priority is higher than the linux task, but lower than all the other HRT Tasks. The IRQ monitor task is waked when a event is inserted. It will be block again when all the events are processed.

4.2.2.3 System call services

There are various system call services provided by the IRQ Manager. Some of them are:

- Change the current PC to an arbitrary address and run it under user mode.
- Allow Linux interrupt handlers to return control back to the xLuna kernel.
- A system call that is used by linux to register the Kernel Interface Table.
- Allow linux to simulate the enable/disable of hardware interrupts.
- Finally a system call that is used by Linux to turn off itself.

4.2.2.4 IRQ manager API

It provides APIs for RTEMS tasks do the following tasks:

- A RTEMS task to boot Linux subsystem.
- Another RTEMS task that determine if linux task is already running.
- Finally, a RTEMS task to kill Linux task.

4.3 Summary

We described in this chapter the OS that is the target of the Formal Verification that is described in the next chapter.

Chapter 5

Formal Verification of IRQ Manager

In this chapter, the verification process will be explained. It is divided in six different sections:

- *Overview* : A general overview of the verification.
- *Functions Verified* : The presentation of the contracts builded and its explanation.
- *Next Steps* : The list of the functions that were not verified.
- *Results* : The results of the proof obligations created.
- *Static Verification* : An analyze of how Static verification helped the construction of the Contracts.
- *Contracts Analyze* : A critic analyze of the contracts.

5.1 Overview

The verification of the IRQ Manager of xLuna was the objective of this project.

The main focus of the Frama-C verification was the safety and functional issues of xLuna IRQ Manager. The initial stage comprised the evaluation and tailoring of the original code: since, at the moment, Frama-C does not support all the C language, some functions had to be changed to resolve some incompatibilities. Other functions were excluded of the verification, since the code present in those functions was completely incompatible with Frama-C. After the initial phase, the next step was to start the analyze of the code and to start to build the contracts necessary to guarantee that the code was reliable.

The main topics covered in the verification were:

- *Pointer dereferencing*: the code of the xLuna IRQ Manager has a significant number of global variables and pointers. As such, pointer dereferencing was one of the most relevant aspects;
- *Arguments*: analyze if the arguments that are in the function are correctly introduced and do not prevent the functions from terminating;
- *Loops Termination*: analyze the body of the loops and build the necessary contracts (loop variants and loop invariants) to prove that each loop terminates and gives the correct result;

Now the contracts implemented will be shown. The description of the functions and the explanation of the contracts will be presented.

5.2 Functions Verified

This section will be organized as follows:

- Each function will have a subsection with its name.
- The description of the function start the subsection.
- The original xLuna code with the contracts will follow the description.
- Next we will explain the lines were the contracts were addicted.
- Finally the number of proof obligations created by the VCGen will be shown.

5.2.1 Function event queue is not empty

Description: The objective of this function is to see if there is any event in the queue.

Function C Code:

```
 1 /*@ requires \valid_range(&event_queue.asyn_events,0,
      NUMBER_OF_ASYN_EVENTS);
 2  ensures \result==0 || \result==1;
 3  */
 4  static inline int event_queue_is_not_empty(void)
 5 {
 6   int i;
 7   if (event_queue_has_sync_event()!=0) return 1;
 8   //@assert NUMBER_OF_ASYN_EVENTS<=(TT_ASYN_MAX - TT_ASYN_MIN + 1);
 9   //@assert NUMBER_OF_ASYN_EVENTS>=0;
10   /*@ loop invariant 0 <= i <= NUMBER_OF_ASYN_EVENTS;
11    @loop variant NUMBER_OF_ASYN_EVENTS-i;
12    @loop invariant NUMBER_OF_ASYN_EVENTS<=(TT_ASYN_MAX - TT_ASYN_MIN +
         1);
13    @loop invariant \forall integer i; 0 <= i < NUMBER_OF_ASYN_EVENTS ==>
          \valid(&event_queue.asyn_events[i]);
14    @*/
15   for (i=0; i<NUMBER_OF_ASYN_EVENTS; i++){
16     if (&event_queue.asyn_events[i].pending) return 1;}
17   return 0;
18 }
```

Listing 5.1: Function event queue is not empty

The contracts included in this function are explained as follows:

- Line 1: This contract is a precondition: the function to finish with success needs that the field *asyn events* of the event queue must be valid between 0 and the number of *asyn events*.

- Line 2: A postcondition that guarantees that the output of the function is either 0 or -1. (It is 1 if the are events in the queue. If not the output is 0.)

- Line 8 and 9: Two assertions that guarantee that the number of *asyn events* is valid.

- Line 10: A loop invariant that says that the integer *i*, is between 0 and the number of asyn events.

- Line 11: A loop variant: it guarantees that the loop will terminate.

- Line 13: It is a loop invariant that guarantees that in the loop all the fields of the *asyn events* of the *event queue* are valid.

The VCGen created 33 proof obligations for this function.

5.2.2 Function event queue pop

Description: This function retires the first event from the queue.

Function C Code:

```
 1 /*@ requires \valid_range(&event_queue.asyn_events,0,
       NUMBER_OF_ASYN_EVENTS);
 2   requires event_to_index(irq->event) <= NUMBER_OF_ASYN_EVENTS;
 3   requires event_to_index(irq->event) >=0;
 4   requires \valid(irq);
 5   */
 6 int event_queue_pop(struct irq_entry *irq)
 7 {
 8     int level;
 9     int ret = -1;
10     int i;
11     //level = sparc_disable_interrupts();
12     if (event_queue_has_sync_event()) {
13         event_queue.sync_event.event = 0;
14         ret = 0;
15     } else {
16         /*@ loop invariant 0 <= i <= NUMBER_OF_ASYN_EVENTS;
17          @loop variant NUMBER_OF_ASYN_EVENTS-i;
18          @loop invariant NUMBER_OF_ASYN_EVENTS<=(TT_ASYN_MAX - TT_ASYN_MIN +
              1);
19          @loop invariant \forall integer i; 0 <= i < NUMBER_OF_ASYN_EVENTS
              ==> \valid(&event_queue.asyn_events[i]);
20          @*/
21         for (i=0; i< NUMBER_OF_ASYN_EVENTS; i++) {
22             if (event_queue.asyn_events[i].pending) {
23                 irq->data = NULL;
24                 irq->event = i + TT_ASYN_MIN;
25                 ret = 0;
26                 break;
27             }
```

```
      }
   }
   //sparc_enable_interrupts(level);
   return ret;
}
```

Listing 5.2: Function event queue pop

The contracts included in this function are explained as follows:

- Line 1: This contract is a precondition: the function to finish with success needs that the field *asyn events* of the event queue must be valid between 0 and the number of *asyn events*.
- Line 2 and 3: These preconditions say that the event on irq is between 0 and the number of *asyn events*.
- Line 4: A precondition that it is required that *irq* pointer is allocated in a safely memory location.
- Line 16: A loop invariant that says that the integer *i*, is between 0 and the number of asyn events.
- Line 17: A loop variant: it guarantees that the loop will terminate.
- Line 19: It is a loop invariant that guarantees that in the loop all the fields of the *asyn events* of the event queue are valid.

The VCGen created 34 proof obligations for this function.

5.2.3 Function isc syscall dispatcher

Description: This routine implements the RTEMS semaphore release directive. It frees a unit to the semaphore associated with ID. If a task was blocked waiting for a unit from this semaphore, then that task will be readied and the unit given to that task. Otherwise, the unit will be returned to the semaphore. *Function C Code*:

```
/*@ requires \valid(isf);
 @ ensures \result ==0 || \result==1;
 @*/
```

```
int isc_syscall_dispatcher(CPU_Interrupt_frame *isf)
{
  rtems_status_code ret=0;
  //@ assert ret>=0 && isf->i1>=0;
  if (isf->i2 == ISC_READ) {
    ret = rtems_message_queue_receive(rtems_to_lx_id ,
     (void *)isf->i1 , isf->i3 ,
     RTEMS_NO_WAIT, RTEMS_NO_TIMEOUT);
    if (ret == RTEMS_SUCCESSFUL) {
      //@ assert ret== RTEMS_SUCCESSFUL;
      isf->i0 = 0;
      rtems_semaphore_release(n_empty_id);
    } else {
      if (ret == RTEMS_UNSATISFIED)
        //@ assert ret== RTEMS_UNSATISFIED;
        isf->i0 = EAGAIN;
      else
        isf->i0 = EIO;
    }
    return 1;
  } else if (isf->i2 == ISC_WRITE) {
    ret = rtems_message_queue_send(lx_to_rtems_id ,
                (void *)isf->i1 , isf->i3);
    if (ret == RTEMS_SUCCESSFUL)
      //@ assert ret== RTEMS_SUCCESSFUL;
      isf->i0 = 0;
    else {
      if (ret == RTEMS_TOO_MANY)
        //@ assert ret== RTEMS_TOO_MANY;
        isf->i0 = EAGAIN;
      else if (ret == RTEMS_INVALID_SIZE)
        //@ assert ret== RTEMS_INVALID_SIZE;
        isf->i0 = EINVAL;
      else
        isf->i0 = EIO;
    }
    return 1;
  } else if (isf->i2 == ISC_GET_NUMBER_PENDING) {
    isf->i0 = 1;
    return 1;
  }
  return 0;
}
```

Listing 5.3: Function isc syscall dispatcher

The contracts included in this function are explained as follows:

- Line 1: A precondition that requires a valid value of *isf*.
- Line 2: This postcondition guarantees that the output of the function will be 0 or 1.
- Line 7: An assertion that says that *ret* will allways equal or bigger than 0 and that *isf i1* will also be equal or bigger than 0.
- Line 13, 18, 28, 32, 35: Those assertions guarantee that the variable *ret* will have the right value when enter on that if statement.

The VCGen created 23 proof obligations for this function.

5.2.4 Remaining Function

The analyze of the remaining functions is in the Appendix A.

5.3 Next Steps

In this section we will show what were the functions that were not verified. So, the unverified functions of xLuna were:

1. *irq do unhandled exception*
2. *irq xluna syscall dispatcher*
3. *irq restore linux context*
4. *irq build linux frame*
5. *irq search linux chain fp*

It was impossible to do the verification of this functions due to incompatibilities of Frama-C with the source code present on that functions. The objective of the Verification was to verify the real source code of IRQ Manager. In those functions there were a lot of lines to remove due to Frama-C incompatibility. So, the solution was to not verify that functions.

5.4 Results

In this section the results of the verification of two files will be presented: the irq.c and the isc.c. This files are from the IRQ Manager of xLuna. The following results will be shown:

- Why prover isc.c results.
- Why prover irq.c results.
- The result of a function of isc.c.
- The result of a function of irq.c.
- The analyze of the results.

5.4.1 ISC.C: Why prover

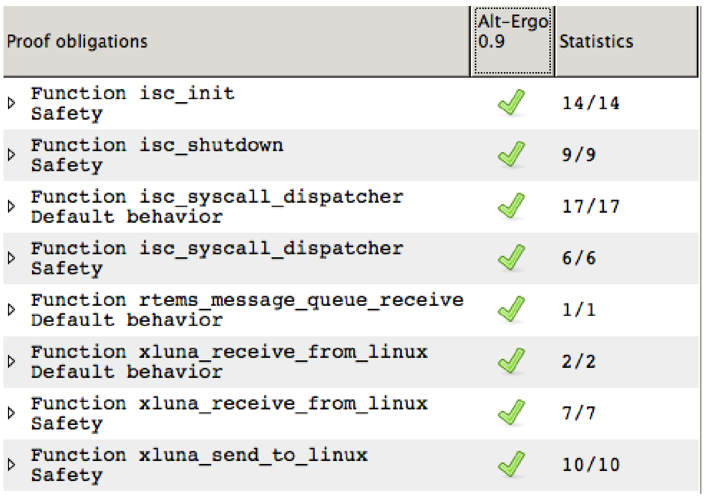
Here we will have a screenshot of the Why platform with the functions and the results of the analyze. So the Figure 5.1 shows the results.

In the total the VCGen created 66 proof obligations.

5.4.2 IRQ.C: Why prover

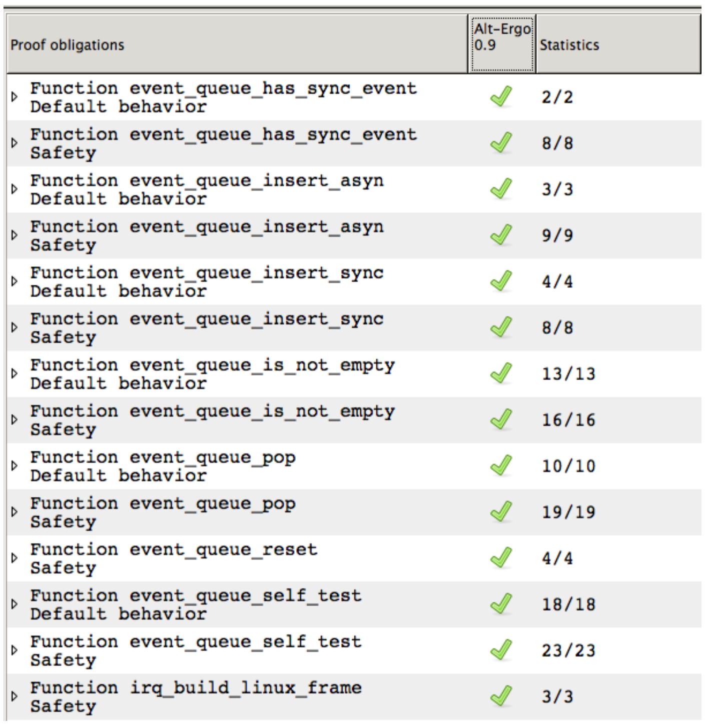
Now a screenshot of the Why platform with the functions and the results of the analyze will be inserted. The Figure 5.2 and Figure 5.3 shows the results.

In total were created 373 proof obligations.



Proof obligations	Alt-Ergo 0.9	Statistics
Function isc_init Safety	✓	14/14
Function isc_shutdown Safety	✓	9/9
Function isc_syscall_dispatcher Default behavior	✓	17/17
Function isc_syscall_dispatcher Safety	✓	6/6
Function rtems_message_queue_receive Default behavior	✓	1/1
Function xluna_receive_from_linux Default behavior	✓	2/2
Function xluna_receive_from_linux Safety	✓	7/7
Function xluna_send_to_linux Safety	✓	10/10

Figure 5.1: ISC



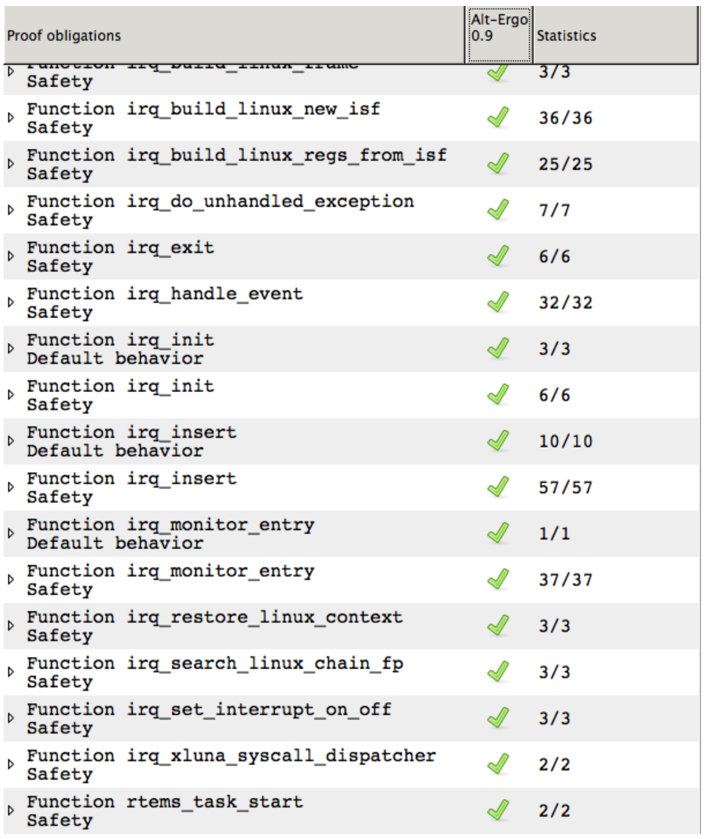
Proof obligations	Alt-Ergo 0.9	Statistics
Function event_queue_has_sync_event Default behavior	✓	2/2
Function event_queue_has_sync_event Safety	✓	8/8
Function event_queue_insert_asyn Default behavior	✓	3/3
Function event_queue_insert_asyn Safety	✓	9/9
Function event_queue_insert_sync Default behavior	✓	4/4
Function event_queue_insert_sync Safety	✓	8/8
Function event_queue_is_not_empty Default behavior	✓	13/13
Function event_queue_is_not_empty Safety	✓	16/16
Function event_queue_pop Default behavior	✓	10/10
Function event_queue_pop Safety	✓	19/19
Function event_queue_reset Safety	✓	4/4
Function event_queue_self_test Default behavior	✓	18/18
Function event_queue_self_test Safety	✓	23/23
Function irq_build_linux_frame Safety	✓	3/3

Figure 5.2: IRQ Why prover results 1/2

5.4.3 ISC.C function: Why prover

In this section a example of a isc.c function and the proof obligations created by the why prover will be shown. In the Figure 5.4 we can see the Why analyze of the *isc call dispatcher* function that is explained in the section 5.2.3.

Looking to the figure 5.4, can be seen that the proof obligations created to this



Proof obligations	Alt-Ergo 0.9	Statistics
Function irq_build_linux_frame Safety	✓	3/3
Function irq_build_linux_new_isf Safety	✓	36/36
Function irq_build_linux_regs_from_isf Safety	✓	25/25
Function irq_do_unhandled_exception Safety	✓	7/7
Function irq_exit Safety	✓	6/6
Function irq_handle_event Safety	✓	32/32
Function irq_init Default behavior	✓	3/3
Function irq_init Safety	✓	6/6
Function irq_insert Default behavior	✓	10/10
Function irq_insert Safety	✓	57/57
Function irq_monitor_entry Default behavior	✓	1/1
Function irq_monitor_entry Safety	✓	37/37
Function irq_restore_linux_context Safety	✓	3/3
Function irq_search_linux_chain_fp Safety	✓	3/3
Function irq_set_interrupt_on_off Safety	✓	3/3
Function irq_xluna_syscall_dispatcher Safety	✓	2/2
Function rtems_task_start Safety	✓	2/2

Figure 5.3: IRQ Why prover results 2/2

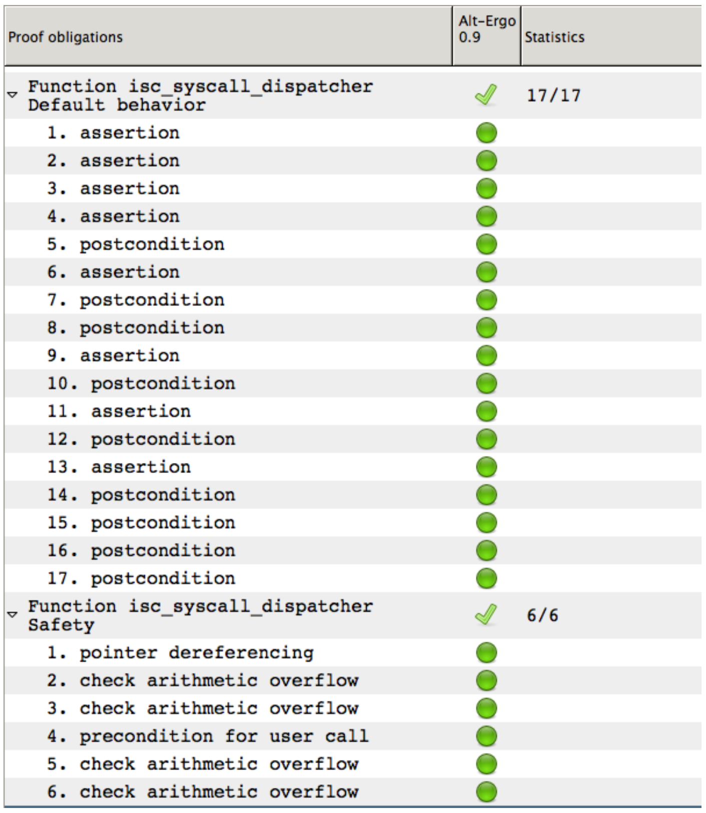
function were all based in assertions and postconditions. If we look to the code presented in the section 5.2.3, we can see that most of the contracts created were assertions. So, we have a lot of proof obligations of assertions. The postconditions proof obligations created were based on the return of the functions. In any *return* it has to check if the postcondition contract is respected. It as also pointers dereferencing and check arithmetic overflow.

On Figure 5.5 we can see that the why prover shows the exact place of the contract. In this case we can see what was the assertion that was verified.

5.4.4 IRQ.C function: Why prover

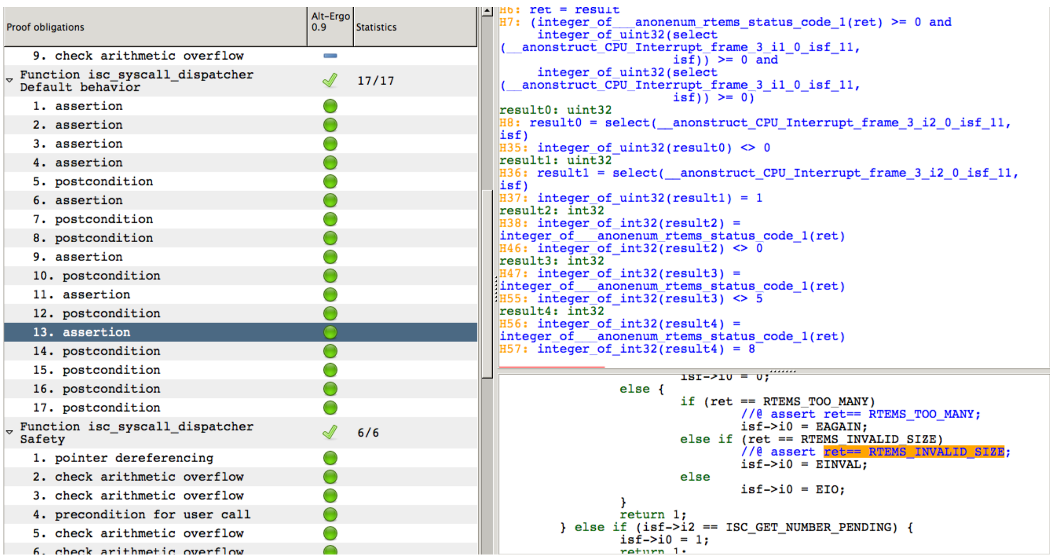
Now the example of a irq.c function. As in the previous section a image with the proof obligations created will be shown. In the Figure 5.6 we can see the Why analyze of the *event queue pop* function that is explained in the section 5.2.2.

In the irq.c example we have a lot of more different proof obligations. It has



Proof obligations	Alt-Ergo 0.9	Statistics
Function isc_syscall_dispatcher Default behavior	✓	17/17
1. assertion	●	
2. assertion	●	
3. assertion	●	
4. assertion	●	
5. postcondition	●	
6. assertion	●	
7. postcondition	●	
8. postcondition	●	
9. assertion	●	
10. postcondition	●	
11. assertion	●	
12. postcondition	●	
13. assertion	●	
14. postcondition	●	
15. postcondition	●	
16. postcondition	●	
17. postcondition	●	
Function isc_syscall_dispatcher Safety	✓	6/6
1. pointer dereferencing	●	
2. check arithmetic overflow	●	
3. check arithmetic overflow	●	
4. precondition for user call	●	
5. check arithmetic overflow	●	
6. check arithmetic overflow	●	

Figure 5.4: ISC function example



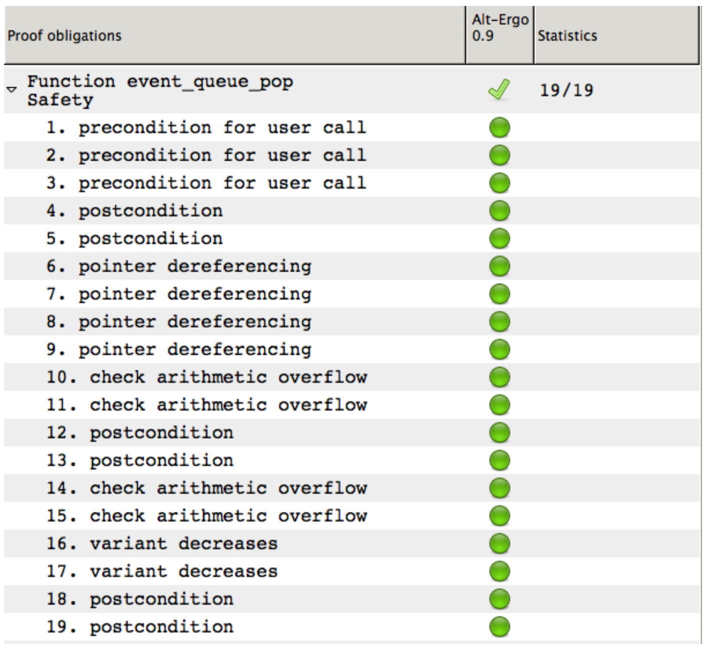
Proof obligations	Alt-Ergo 0.9	Statistics
9. check arithmetic overflow	▬	
Function isc_syscall_dispatcher Default behavior	✓	17/17
1. assertion	●	
2. assertion	●	
3. assertion	●	
4. assertion	●	
5. postcondition	●	
6. assertion	●	
7. postcondition	●	
8. postcondition	●	
9. assertion	●	
10. postcondition	●	
11. assertion	●	
12. postcondition	●	
13. assertion	●	
14. postcondition	●	
15. postcondition	●	
16. postcondition	●	
17. postcondition	●	
Function isc_syscall_dispatcher Safety	✓	6/6
1. pointer dereferencing	●	
2. check arithmetic overflow	●	
3. check arithmetic overflow	●	
4. precondition for user call	●	
5. check arithmetic overflow	●	
6. check arithmetic overflow	●	

```
H6: ret = result
H7: (integer_of___anonenum_rtems_status_code_1(ret) >= 0 and
     integer_of_uint32(select
(__anonstruct_CPU_Interrupt_frame_3_i1_0_isf_11,
                  isf)) >= 0 and
     integer_of_uint32(select
(__anonstruct_CPU_Interrupt_frame_3_i1_0_isf_11,
                  isf)) >= 0)
result0: uint32
H8: result0 = select(__anonstruct_CPU_Interrupt_frame_3_i2_0_isf_11,
isf)
H35: integer_of_uint32(result0) <> 0
result1: uint32
H36: result1 = select(__anonstruct_CPU_Interrupt_frame_3_i2_0_isf_11,
isf)
H37: integer_of_uint32(result1) = 1
result2: int32
H38: integer_of_int32(result2) =
integer_of___anonenum_rtems_status_code_1(ret)
H46: integer_of_int32(result2) <> 0
result3: int32
H47: integer_of_int32(result3) =
integer_of___anonenum_rtems_status_code_1(ret)
H55: integer_of_int32(result3) <> 5
result4: int32
H56: integer_of_int32(result4) =
integer_of___anonenum_rtems_status_code_1(ret)
H57: integer_of_int32(result4) = 8
```

```
                        isf->i0 = 0;
                else {
                        if (ret == RTEMS_TOO_MANY)
                                //@ assert ret== RTEMS_TOO_MANY;
                                isf->i0 = EAGAIN;
                        else if (ret == RTEMS_INVALID_SIZE)
                                //@ assert ret== RTEMS_INVALID_SIZE;
                                isf->i0 = EINVAL;
                        else
                                isf->i0 = EIO;
                }
                return 1;
        } else if (isf->i2 == ISC_GET_NUMBER_PENDING) {
                isf->i0 = 1;
                return 1;
```

Figure 5.5: ISC function example 2

preconditions checks for the functions called inside the function. Because this functions has a loop it also checks if the variant decreases in the loops. The *return* of the function is also checked in various proof obligations. Four *check arithmetic overflow*

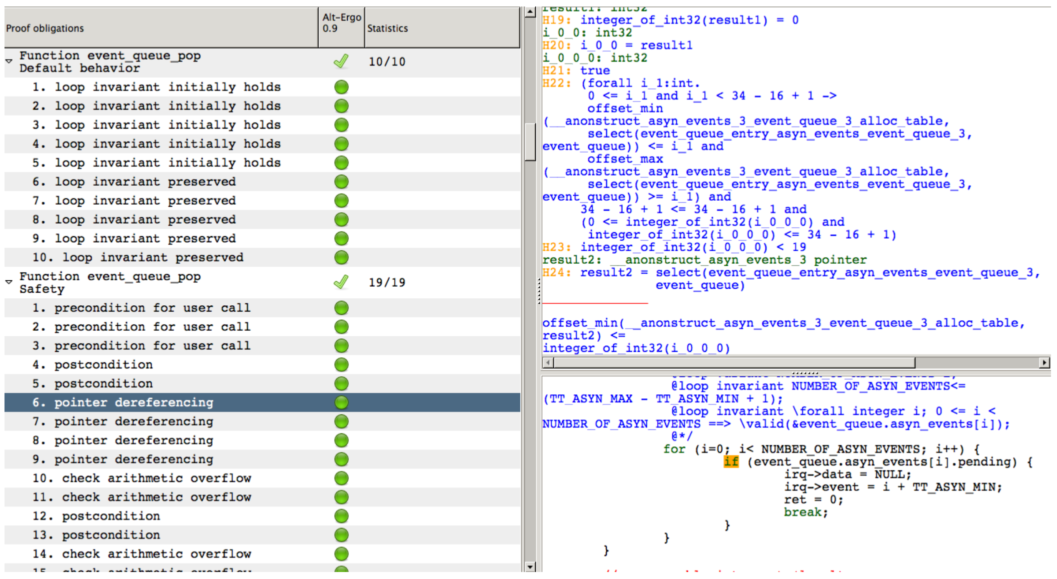


Proof obligations	Alt-Ergo 0.9	Statistics
Function event_queue_pop Safety	✓	19/19
1. precondition for user call	●	
2. precondition for user call	●	
3. precondition for user call	●	
4. postcondition	●	
5. postcondition	●	
6. pointer dereferencing	●	
7. pointer dereferencing	●	
8. pointer dereferencing	●	
9. pointer dereferencing	●	
10. check arithmetic overflow	●	
11. check arithmetic overflow	●	
12. postcondition	●	
13. postcondition	●	
14. check arithmetic overflow	●	
15. check arithmetic overflow	●	
16. variant decreases	●	
17. variant decreases	●	
18. postcondition	●	
19. postcondition	●	

Figure 5.6: IRQ function example

proof obligations are also created. The preconditions are also verified.

In this function we had twenty nine proof obligations. One of them was a pointer dereferencing shown on the Figure 5.7. That pointer dereferencing was the verification of the pending field of the struct *asyn events*. With the contracts created the verification of this proof obligations was a success.



Proof obligations	Alt-Ergo 0.9	Statistics
Function event_queue_pop Default behavior	✓	10/10
1. loop invariant initially holds	●	
2. loop invariant initially holds	●	
3. loop invariant initially holds	●	
4. loop invariant initially holds	●	
5. loop invariant initially holds	●	
6. loop invariant preserved	●	
7. loop invariant preserved	●	
8. loop invariant preserved	●	
9. loop invariant preserved	●	
10. loop invariant preserved	●	
Function event_queue_pop Safety	✓	19/19
1. precondition for user call	●	
2. precondition for user call	●	
3. precondition for user call	●	
4. postcondition	●	
5. postcondition	●	
6. pointer dereferencing	●	
7. pointer dereferencing	●	
8. pointer dereferencing	●	
9. pointer dereferencing	●	
10. check arithmetic overflow	●	
11. check arithmetic overflow	●	
12. postcondition	●	
13. postcondition	●	
14. check arithmetic overflow	●	

```
result1: int32
H19: integer_of_int32(result1) = 0
i_0_0: int32
H20: i_0_0 = result1
i_0_0_0: int32
H21: true
H22: (forall i_1:int.
       0 <= i_1 and i_1 < 34 - 16 + 1 ->
       offset_min
(__anonstruct_asyn_events_3_event_queue_3_alloc_table,
       select(event_queue_entry_asyn_events_event_queue_3,
event_queue)) <= i_1 and
       offset_max
(__anonstruct_asyn_events_3_event_queue_3_alloc_table,
       select(event_queue_entry_asyn_events_event_queue_3,
event_queue)) >= i_1) and
      34 - 16 + 1 <= 34 - 16 + 1 and
      (0 <= integer_of_int32(i_0_0_0) and
       integer_of_int32(i_0_0_0) <= 34 - 16 + 1)
H23: integer_of_int32(i_0_0_0) < 19
result2: __anonstruct_asyn_events_3 pointer
H24: result2 = select(event_queue_entry_asyn_events_event_queue_3,
                event_queue)

offset_min(__anonstruct_asyn_events_3_event_queue_3_alloc_table,
result2) <=
integer_of_int32(i_0_0_0)
```

```
                  @loop invariant NUMBER_OF_ASYN_EVENTS<=
(TT_ASYN_MAX - TT_ASYN_MIN + 1);
                  @loop invariant \forall integer i; 0 <= i <
NUMBER_OF_ASYN_EVENTS ==> \valid(&event_queue.asyn_events[i]);
                  @*/
                 for (i=0; i< NUMBER_OF_ASYN_EVENTS; i++) {
                         if (event_queue.asyn_events[i].pending) {
                                 irq->data = NULL;
                                 irq->event = i + TT_ASYN_MIN;
                                 ret = 0;
                                 break;
                         }
                 }
         }
```

Figure 5.7: IRQ function example 2

5.4.5 Analyzies of the Proof Obligations

Looking into the past two sections of this chapter we can see that were created 66 proof obligations in the file isc.c and 373 proof obligations in the irq.c. On the total were created 439 proof obligations by the VCGen. The automatic prover, the Alt-ergo, proved all of those obligations with success.

5.5 Static Verification

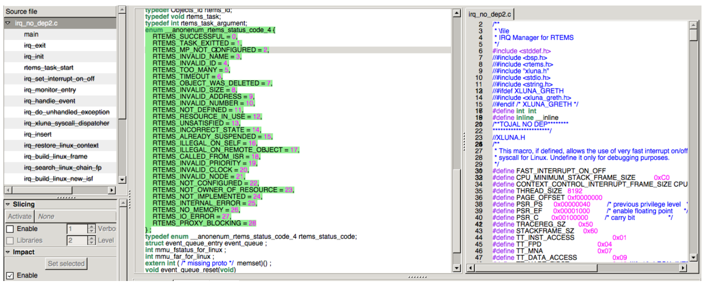
Related to the static verification, we used three plug-ins available in Frama-C. They were:

1. Value analysis plug-in.

2. Occurrence analysis plug-in.

3. Sparecode plug-in.

On the Figure 5.8 we can see a little example of the static verification in Frama-C.



```
typedef Objects_Id rtems_id;
typedef void rtems_task;
typedef int rtems_task_argument;
enum __anonenum_rtems_status_code_4 {
   RTEMS_SUCCESSFUL = 0,
   RTEMS_TASK_EXITTED = 1,
   RTEMS_MP_NOT_CONFIGURED = 2,
   RTEMS_INVALID_NAME = 3,
   RTEMS_INVALID_ID = 4,
   RTEMS_TOO_MANY = 5,
   RTEMS_TIMEOUT = 6,
   RTEMS_OBJECT_WAS_DELETED = 7,
   RTEMS_INVALID_SIZE = 8,
   RTEMS_INVALID_ADDRESS = 9,
   RTEMS_INVALID_NUMBER = 10,
   RTEMS_NOT_DEFINED = 11,
   RTEMS_RESOURCE_IN_USE = 12,
   RTEMS_UNSATISFIED = 13,
   RTEMS_INCORRECT_STATE = 14,
   RTEMS_ALREADY_SUSPENDED = 15,
   RTEMS_ILLEGAL_ON_SELF = 16,
   RTEMS_ILLEGAL_ON_REMOTE_OBJECT = 17,
   RTEMS_CALLED_FROM_ISR = 18,
   RTEMS_INVALID_PRIORITY = 19,
   RTEMS_INVALID_CLOCK = 20,
   RTEMS_INVALID_NODE = 21,
   RTEMS_NOT_CONFIGURED = 22,
   RTEMS_NOT_OWNER_OF_RESOURCE = 23,
   RTEMS_NOT_IMPLEMENTED = 24,
   RTEMS_INTERNAL_ERROR = 25,
   RTEMS_NO_MEMORY = 26,
   RTEMS_IO_ERROR = 27,
   RTEMS_PROXY_BLOCKING = 28
};
typedef enum __anonenum_rtems_status_code_4 rtems_status_code;
struct event_queue_entry event_queue ;
int mmu_fstatus_for_linux ;
int mmu_far_for_linux ;
extern int ( /* missing proto */  memset)() ;
void event_queue_reset(void)
```

```
/**
 * \file
 * IRQ Manager for RTEMS
 */
#include <stddef.h>
//#include <bsp.h>
//#include <rtems.h>
//#include "xluna.h"
//#include <stdio.h>
//#include <string.h>
//#ifdef XLUNA_GRETH
//#include <xluna_greth.h>
//#endif /* XLUNA_GRETH */
#define int  int
#define inline __inline
/**TOJAL NO DEP********
**********************/
//XLUNA.H
/**
 * This macro, if defined, allows the use of very fast interrupt on/off
 * syscall for Linux. Undefine it only for debugging purposes.
 */
#define FAST_INTERRUPT_ON_OFF
#define CPU_MINIMUM_STACK_FRAME_SIZE        0xC0
#define CONTEXT_CONTROL_INTERRUPT_FRAME_SIZE CPU
#define THREAD_SIZE   8192
#define PAGE_OFFSET  0xf0000000
#define PSR_PS      0x00000040        /* previous privilege level
#define PSR_EF      0x00001000        /* enable floating point     */
#define PSR_C      0x00100000        /* carry bit                */
#define TRACEREG_SZ      0x50
#define STACKFRAME_SZ    0x60
#define TT_INST_ACCESS                  0x01
#define TT_FPD                          0x04
#define TT_MNA                          0x07
#define TT_DATA_ACCESS                  0x09
```

Figure 5.8: Frama-C Static Verification

The utilization of this plug-ins were very useful in order to test how the variables behave in all the functions. It also provides with the confirmation that there were not any dead code in the files analyzed. It helped the building of the contracts because with the results of the plug-ins we were able to have a better idea of the behavior of all variables in the functions.

5.6 Contracts Analysis

Looking into the results of the Verification we can say that the contracts added to the code were sufficient to formally verify the functions of IRQ Manager source code compatible with Frama-C. In the construction of the contracts we choose this contracts because we wanted to do a totally automatic verification. So all of the verification was orientated to building contracts that could be verified by an automatic theorem prover. A human-directed proof was totally excluded. So, we excluded any type of contract that couldn't be verified by an automatic theorem prover.

5.7 Summary

In this chapter we present how the Formal Verification of the IRQ Manager was done and what results were achieved. Following the conclusions will be shown.

Chapter 6

Conclusion and Future Work

6.1 Frama-C Toolset

After this work we are able to do an analyze of the platform Frama-C. It is a platform that is in developing and in the last months has been in constant evolution. In this moment it is a platform that is good to work with. The integration of the ACSL, Jessie and the why platform makes Frama-C a very good platform to work in the verification of programs. However it has some problems that the development team needs to improve. Some of those problems are:

- Frama-C has a clearly limitation in supporting pointers. And that is a problem when looking into the code found in an OS module.

- The error messages are not clear. Sometimes it does not says what and where the problem is.

- As the implementation is not stable, it crashes frequently.

- Some of the contracts cause crashes in Frama-C or on its plug-ins.

Despite this errors the verification of a good part of the IRQ Manager was possible. So, ignoring those problems, I can say that Frama-C is a very suitable tool to execute the formal verification of C code programs. The help that the static analyzers gives makes them a very interesting tool. Integrating ACSL, deductive analyzers and static analyzers makes Frama-C one of the most promissing tools in Formal Verification.

6.2 Formal Verification

With Frama-C, in the functions that we analyze, the results on the automatic prover were a success, and all the proof obligations of those function were checked with success. As said before, some code of a few functions were removed because of Frama-c incompatibilities, and other functions were not analyzed because of completely incompatibility with Frama-C. We have verified about 80% of the IRQ Manager code. So as the Frama-C development proceed it will be possible to reintroduce the code removed and to analyze the functions that we could not analyze with the actual version of Frama-C. After the work done in the verification of the xLuna IRQ Manager we should be able to proceed to other xLuna modules. As long as the xLuna modules are not too incompatible with the actual version of Frama-C the verification of those modules may be possible. However with the evolution of Frama-C it may be possible to verify all of xLuna modules.

6.3 Future Work

As said in the last paragraph of the previous section, the complete verification process of xLuna is the ultimate goal. To do that with Frama-C, it has to continue developing into the phase that all the C Code is compatible with it.

When that happens the rest of the IRQ Manager code can be reintroduce and can be verified. After that the other modules of xLuna can also be verified.

The ultimate goal of this type of verification is to obtain the highest standard of safety, explained in the Introduction, EAL7. After that, the example of the seL4 verification can be pursuit and try to achieve a *fully formally verified for functional correctness* xLuna.

As said before the D0-178B evaluation system is the most used in the avionics and space industries. So, the final goal of a formal verification of xLuna is to achieve a DAL-A. It will definitely increase the value of xLuna when compared with rival OS.

Appendix A

Appendix

A.1 IRQ Manager Annotated Code

A.1.1 Function event queue insert asyn

Description: The objective of this function is to insert an asynchronous event into the queue.

Function C Code:

```
/*@ requires \valid_range(&event_queue.asyn_events,0,
    NUMBER_OF_ASYN_EVENTS);
 requires event_to_index(irq->event) <= NUMBER_OF_ASYN_EVENTS;
 requires event_to_index(irq->event) >=0;
 requires \valid(irq);
 ensures \result==0 ;
 */
int event_queue_insert_asyn(struct irq_entry *irq)
{
  int *pending;
  int level;
  ASSERT(is_asyn_trap(irq->event));
  //@ assert event_to_index(irq->event) <= NUMBER_OF_ASYN_EVENTS;
  //@ assert event_to_index(irq->event) >= 0;
  //@ ensures pending == &(event_queue.asyn_events[event_to_index(irq->
      event)].pending) + 1;
  pending = &(event_queue.asyn_events[event_to_index(irq->event)].
      pending);
16   //level = sparc_disable_interrupts();
17   pending = pending + 1;
18   //sparc_enable_interrupts(level);
19   return 0;
20 }
```

Listing A.1: Function event queue insert asyn

The contracts included in this function are explained as follows:

- Line 1: This contract is a precondition: the function to finish with success needs that the field *asyn events* of the *event queue* must be valid between 0 and the number of *asyn events*.

- Line 2 and 3: These preconditions say that the event on irq is between 0 and the number of *asyn events*.

- Line 4: A precondition that it's required that irq pointer is allocated in a safely memory location.

- Line 14: A postcondition, placed here, because it makes it easier to analyze because it's right before the code line that is being analyzed. it ensures that the pending will be equal to the field pending in the *event queue* struct.

The VCGen created 13 proof obligations for this function.

A.1.2 Function event queue insert sync

Description: The objective of this function is to insert a synchronous event in to the queue.

Function C Code:

```
/*@ requires \valid(irq);
  requires(irq->data!=NULL);
 ensures \result==0 || \result==(-1);
 */
int event_queue_insert_sync(struct irq_entry *irq)
{
 7    int level;
 8    ASSERT(irq->event);
 9    if (event_queue_has_sync_event()) return -1;
10    //@ assert \valid(irq);
11    //@ ensures event_queue.sync_event == *irq;
12    event_queue.sync_event = *irq;
13    return 0;
14 }
```

Listing A.2: Function event queue insert sync

The contracts included in this function are explained as follows:

- Line 1: This contract is a precondition: the function to finish with success needs a valid *irq*.
- Line 2: Also a precondition. In this contract we say that the *data field* in the irq cannot be null.
- Line 3: A postcondition that guarantees that the output of the function is either 0 or -1. (It is 0 if the event is inserted; if not the output is -1.)
- Line 8: This ASSERT is not part of the contracts that we build. Is an original xLuna C statement.
- Line 10: An assertion. It strengthens the precondition on line 1. It is right before the place where is absolutely necessary that *irq* is not NULL.
- Line 11: The postcondition of the function.

The VCGen created 15 proof obligations for this function.

A.1.3 Function event queue self test

Description: This function will test if the queue has any event, synchronous or asynchronous.

Function C Code:

```
void event_queue_self_test(void)
  {
    irq_entry irq;
    int i;
    event_queue_reset();
    ASSERT(event_queue_is_not_empty() == 0);
    ASSERT(event_queue_is_not_full());
    ASSERT(event_queue_has_sync_event() == 0);
    ASSERT(event_queue_pop(&irq) == -1);

    i = 0;
    /*@ loop invariant 0 <= i <= NUMBER_OF_ASYN_EVENTS;
     @loop variant NUMBER_OF_ASYN_EVENTS-i;
     @loop invariant NUMBER_OF_ASYN_EVENTS<=(TT_ASYN_MAX - TT_ASYN_MIN +
          1);
     @loop invariant \forall integer i; 0 <= i < NUMBER_OF_ASYN_EVENTS
         ==> \valid(&irq);
     @*/
    while (i<NUMBER_OF_ASYN_EVENTS) {
      i=i+1;
      irq.event = i;
      irq.data = NULL;
      //@requires \valid(&irq);
      //@requires irq.event<= NUMBER_OF_ASYN_EVENTS;
      //@requires irq.event>=0;
      if ((&irq)!=NULL) break;
    }
    ASSERT(i == MAX_EVENTS);
    ASSERT(0==event_queue_insert_sync(&irq));
    ASSERT(-1==event_queue_insert_sync(&irq));
    ASSERT(-1==event_queue_insert_asyn(&irq));
    irq.event = 0;
    ASSERT(0==event_queue_pop(&irq));
    ASSERT(irq.event == MAX_EVENTS);

    /*@ loop invariant 1 <= i <= MAX_EVENTS;
     @loop variant MAX_EVENTS-i;
     @loop invariant MAX_EVENTS==10;
     @*/
    for (i=1; i<MAX_EVENTS; i++) {
```

```
39        ASSERT(0==event_queue_pop(&irq));
40        ASSERT(irq.event == i);
41      }
42      ASSERT(event_queue_pop(&irq) == -1);
43 }
```

Listing A.3: event queue self test

The contracts included in this function are explained as follows:

- Line 12: A loop invariant that says that the integer *i* is between 0 and the number of *asyn events*.
- Line 13: A loop variant: it guarantees that the loop will terminate.
- Line 15: It's a loop invariant that guarantees that in the loop all the fields of the *asyn events* of the *event queue* are valid.
- Line 21: this precondition requires a valid *irq*.
- Line 22 and 23: These preconditions say that the event on irq is between 0 and the number of *asyn events*.
- Line 34: A loop invariant to test the comportment of the variable *i* in the loop.
- Line 35: The loop variant in this loop. Guarantees that the loop will terminate.
- Line 36: it's a loop invariant that says that in this loop the variable *MAX EVENTS* is equal to 10.
- As said before all the ASSERT are not part of the contracts that we build. Is an original xLuna C statement.

The VCGen created 44 proof obligations for this function.

A.1.4 Function irq build linux regs from isf

Description: This function creates the linux registers from the CPU Interrupt frame.

Function C Code:

```
/*@ requires \valid(old_isf);
 requires \valid(regs);
 */
static void irq_build_linux_regs_from_isf(struct pt_regs *regs,
            CPU_Interrupt_frame *old_isf,
            int trap)
{
  regs->psr = old_isf->psr;
  if (IS_KERNEL(old_isf->pc))
    regs->psr = SPARC_PSR_PS_MASK; /* fake kernel mode */
  else
    (0 == (regs->psr && SPARC_PSR_PS_MASK));
  /* pc/npc depends on RTEMS "asyn/sync" trap type ... */
  if (is_asyn_trap(trap)) {
    regs->pc  = old_isf->pc;
    regs->npc = old_isf->npc;
  } else {
    /* rtems already moved npc to isf->pc */
    regs->pc  = old_isf->npc;
    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_I7] = old_isf->i7;
}
```

Listing A.4: Function irq build linux regs from isf

The contracts included in this function are explained as follows:

- Line 1 and 2: Two preconditions that declare that a valid *regs* and a valid *old isf* are required to guarantee that the function terminates with success. As this function only makes access to fields of those variables having valid *regs* and valid *old isf* is the only condition necessary.

The VCGen created 35 proof obligations for this function.

A.1.5 Function irq build linux new isf

Description: In this function a new CPU Interrupt frame will be create.

Function C Code:

```
/*@ requires \valid(old_isf);
 requires \valid(regs);
 requires \valid(new_isf);
 requires trap>=0;
 */
static void irq_build_linux_new_isf(CPU_Interrupt_frame *new_isf,
          struct pt_regs *regs,
          CPU_Interrupt_frame *old_isf,
          int trap)
{
  //@ requires (new_isf->pc) >= 0;
  //@ ensures new_isf->i1 == 1;
  new_isf->npc = new_isf->pc;
  new_isf->i0 = trap;
  new_isf->i1 = 1;
  if (trap == TT_MNA) { /* we need to get the fault type and fault addr
     */
  } else if (trap == TT_DATA_ACCESS) {
     int fstatus, faddr;
    //@ ensures new_isf->i2 == fstatus;
    //@ ensures new_isf->i3 == fstatus;
    //@ ensures new_isf->i4 == faddr;
    faddr = mmu_far_for_linux;
    fstatus = mmu_fstatus_for_linux >> 6;
    new_isf->i2 = fstatus; //    Data = 0; Instrc = 1
25      new_isf->i3 = fstatus; //    Read = 0; Write = 1
26      new_isf->i4 = faddr;
27      if (IS_RTEMS_KERNEL(old_isf->pc)) {
28        DEBUG_puts("Page fault in RTEMS/xLuna!");
29        dump_isf(old_isf);
30      }
31    } else if (trap == TT_INST_ACCESS) {
32       int fstatus, faddr;
33
34      //@ ensures new_isf->i2 == 1;
35      //@ ensures new_isf->i3 == 0;
36      //@ ensures new_isf->i4 == faddr;
37      faddr = mmu_far_for_linux;
38      fstatus = mmu_fstatus_for_linux >> 6;
39      new_isf->i2 = 1; //      text_fault = 1 -> Instruction
40      new_isf->i3 = 0; //      write = 0 -> read
41      new_isf->i4 = faddr;
42      if (IS_RTEMS_KERNEL(old_isf->pc)) {
43                              DEBUG_puts("Page fault in RTEMS/xLuna!");
44                              dump_isf(old_isf);
45                              ASSERT(0);
46      }
47    }
48
49    (concurrent_linux_handlers < MAX_CONCURRENT_LINUX_HANDLERS);
50    ASSERT(concurrent_linux_handlers=concurrent_linux_handlers+1);
51    new_isf->i7 = concurrent_linux_handlers;
52  }
```

Listing A.5: Function irq build linux new isf

The contracts included in this function are explained as follows:

- Line 1, 2 and 3: These preconditions require that the pointers *old isf*, *regs* and *new isf* are valid.

- Line 4: A precondition. These precondition was built based on the xLuna properties that require that the variable *trap* must be equal or bigger than 0.

- Line 11: It's also a xLuna property that the field *i1* of the *CPU Interrupt frame* must be equal or bigger than 0.

- Line 12: A postcondition.
- Line 19, 20 and 21: These postconditions guarantee that if the program reach this point than the fields *i2* and *i3* of the *new isf* will be *fstatus* and the field *i4* will be equal to *faddr*.
- Line 34, 35 and 36: These three postconditions ensures that if the program enters this if statement then the field *i2* will be 1, the field *i3* will be 0 and the field *i4* will be equal to *faddr*.
- As said before all the ASSERT are not part of the contracts that we build. Is an original xLuna C statement.

The VCGen created 48 proof obligations for this function.

A.1.6 Function irq insert

Description: This function is used to insert a new Interrupt Request in the queue.

Function C Code:

```
1  /*@ requires sync==SYNC_EVENT || sync==ASYN_EVENT;
2   requires event>=0;
3   ensures \result == RTEMS_SUCCESSFUL || \result == RTEMS_TOO_MANY;
4   */
5  rtems_status_code irq_insert(int event, void *data, int sync)
6  {
7    irq_entry irq;
8    int ret;
9    static int queue_full_count;
10   //@ ensures irq.data==data && irq.event==event;
11   irq.event = event;
12   irq.data = data;
13   if (!LINUX_IS_RUNNING) return RTEMS_TOO_MANY;
14   if (sync == SYNC_EVENT) {
15     if (-1 == (ret=event_queue_insert_sync(&irq))) {
16       ASSERT(0);
17     }
18   } else {
19     ASSERT(sync == ASYN_EVENT);
20      if (-1 != (ret=event_queue_insert_sync(&irq))) {
21        if (queue_full_count > 100) {
22          ASSERT(0);
23          DEBUG_puts("Queue blocked?");
24        }
25      } else
26        queue_full_count = 0;
27    }
28    rtems_task_resume(irq_monitor_id);
29    if(ret==0) return RTEMS_SUCCESSFUL;
30    else return RTEMS_TOO_MANY;
31  }
```

Listing A.6: Function irq insert

The contracts included in this function are explained as follows:

- Line 1: A precondition that says that the variable sync must be *SYNC EVENT* or *ASYN EVENT*. This is because the variable sync will determine if the event is a synchronous or asynchronous event. *SYNC EVENT* and *ASYN EVENT* are both global variables of the IRQ Manager.
- Line 2: It's a precondition that requires that the variable *event* must be equal or bigger than 0.
- Line 3: A postcondition that guarantees that the output of the function will be *RTEMS SUCCESSFUL* or *RTEMS TOO MANY*. Those are two global variables.
- Line 10: That's a postcondition that guarantees that the fields *data* and *event* of the *irq* will be respectively equal to *data* and *event*.
- As said before all the ASSERT are not part of the contracts that we build. Is an original xLuna C statement.

The VCGen created 84 proof obligations for this function.

A.1.7 Function irq handle event

Description: In this function the event will be treated.

Function C Code:

```
/*@ requires event==TT_UART_FIRST || event==TT_TIMER || event==TT_ASYN_CLEAR_WINDOW || event==TT_ISC_RTEMS_TO_LX ||
@ event==TT_ISC_LX_TO_RTEMS || event==TT_GRETH || 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;
@*/
static void irq_handle_event(int event, void *data)
{
  switch(event) {
    case TT_UART_FIRST:
    case TT_TIMER:
    case TT_ASYN_CLEAR_WINDOW:
    case TT_ISC_RTEMS_TO_LX: //evento inserido na queue pelo isc do rtems
    case TT_ISC_LX_TO_RTEMS: //evento inserido na queue pelo isc do rtems
    case TT_GRETH:
      //@ ensures linux_hardware_handler_in_progress == 1;
      linux_hardware_handler_in_progress = 1;
    case TT_LINUX_SYSCALL:
    case TT_SYNC_KILL_USER:
    case TT_SYNC_CLEAR_WINDOW:
    case TT_MNA:
    case TT_FPD:
    case TT_DATA_ACCESS:
    case TT_INST_ACCESS:
      irq_build_linux_frame(event, linux_isf);
      //@ ensures linux_isf == NULL;
      linux_isf = NULL;
      rtems_task_suspend(RTEMS_SELF);
      break;
    case TT_XLUNA_SYSCALL:
      irq_xluna_syscall_dispatcher(data);
      break;
    case TT_UNHANDLED_EXCEPTION:
      irq_do_unhandled_exception(data);
      break;
35    }
36  }
```

Listing A.7: Function irq handle event

The contracts included in this function are explained as follows:

- Line 1, 2, 3 and 4: These are all precondition related to the switch case in the program. To run well the function needs that the *event* field is one of the cases of switch.

- Line 15 and 25: They are postconditions that guarantees that if the event is that then those attribution are ensured.

The VCGen created 50 proof obligations for this function.

A.1.8 Function irq monitor entry

Description: This function is used to assess if the IRQ queue is empty or not.

Function C Code:

```
//@ requires MAX_EVENTS == 10;
static rtems_task irq_monitor_entry(rtems_task_argument ignored)
{
  irq_entry irq;
  int events_handled=0;
    //@loop variant MAX_EVENTS - events_handled;
    while(MAX_EVENTS>events_handled) {
      if (event_queue_has_sync_event()) {
        //@ invariant events_handled <= MAX_EVENTS;
        events_handled++;
        break;
        ASSERT(irq_handle_event(irq.event, irq.data));
      } else if (LINUX_INTERRUPT_ENABLED) {
        ASSERT(irq_handle_event(irq.event, irq.data));
      } else {
        rtems_task_suspend(RTEMS_SELF);
      }
      events_handled++;
19       }
20       rtems_task_suspend(RTEMS_SELF);
21 }
```

Listing A.8: Function irq monitor entry

The contracts included in this function are explained as follows:

- Line 1: A precondition that requires that the variable *MAX EVENTS* is equal to 10. That is a requirement of the IRQ Manager.
- Line 6: It's a loop variant that guarantees that the loop ends.
- Line 9: A invariant to guarantee that the variable *events handled* don't go overflow.

The VCGen created 46 proof obligations for this function.

A.1.9 Function irq set interrupt on off

Description: This function is used to change if Linux is to be interrupted or not.

Function C Code:

```
1  /*@ requires on_off==0 || on_off==1;
2    ensures \result == ret;
3  */
4  int irq_set_interrupt_on_off(int on_off)
5  {
6    int ret = linux_interrupt_enabled;
7    // ensures linux_interrupt_enabled == on_off;
8    linux_interrupt_enabled = on_off;
9  #if 0
10   if (linux_interrupt_enabled) DEBUG_puts_no_cr("+");
11   else DEBUG_puts_no_cr("-");
12 #endif
13   return ret;}
```

Listing A.9: Function irq set interrupt on off

The contracts included in this function are explained as follows:

- Line 1: A precondition that requires that the variable *on off* must be 0 or 1. It's 0 if the interruptor is enabled. 1 if it is disabled.

- Line 2: A postcondition that guarantees that the output of the function is the variable *ret*.

- Line 7: A postcondition that says that the variable *linux interrupt enabled* will be equal to the variable *on off*.

The VCGen created 4 proof obligations for this function.

A.1.10 Function xluna send to linux

Description: Send a message to Linux. It consists on 3 main steps:

- Obtain N EMPTY semaphore to check how many slots are available.

- Write on the message queue.

- Add and IRQ to signal Linux's N FULL semaphore.

Function C Code:

```
/*@ requires (opt_set==RTEMS_WAIT) || (opt_set==RTEMS_NO_WAIT);
 @ requires size>=0;
 @ requires (timeout>0) || (timeout==RTEMS_NO_TIMEOUT);
 @*/
rtems_status_code xluna_send_to_linux(void *message, rtems_unsigned32 size,
        rtems_unsigned32 opt_set, rtems_interval timeout)
{
  if (size > ISC_MAX_MSG_SIZE) return RTEMS_INVALID_SIZE;
  /* 1-Obtain N_EMPTY semaphore to check how many slots are available */
  CHECK_STATUS_CODE(rtems_semaphore_obtain(n_empty_id, opt_set, timeout));
  /* 2-Write on the message queue */
  CHECK_STATUS_CODE(rtems_message_queue_send(rtems_to_lx_id,
                message, size));
  //3-Add an IRQ to signal Linux's N_FULL semaphore
    irq_insert(TT_ISC_RTEMS_TO_LX, ASYN_EVENT);
  return RTEMS_SUCCESSFUL;
17 }
```

Listing A.10: Function xluna send to linux

The contracts included in this function are explained as follows:

- Line 1: A precondition that requires that the variable *opt set* must be the global variable *RTEMS WAIT* or *RTEMS NO WAIT*.
- Line 2: Precondition that says that the size of the message must be equal or bigger that 0.
- Line 3: This precondition requires that the variable *timeout* must be bigger than 0 or equal to the global variable *RTEMS NO TIMEOUT*.

The VCGen created 10 proof obligations for this function.

A.1.11 Function xluna receive from linux

Description: Receives a message from Linux. It consists on 3 main steps:

- Obtain N FULL semaphore.
- Get the message from the message queue.
- Add and IRQ to signal Linux's N EMPTY semaphore.

Function C Code:

```
 1 /*@ requires (opt_set==RTEMS_WAIT) || (opt_set==RTEMS_NO_WAIT);
 2  @ requires size>=0;
 3  @ requires (timeout>0) || (timeout==RTEMS_NO_TIMEOUT);
 4  @ ensures \result==RTEMS_SUCCESSFUL;
 5  @*/
 6 rtems_status_code xluna_receive_from_linux(void *buffer,
      rtems_unsigned32 size,
 7                 rtems_unsigned32 opt_set,
 8                 rtems_interval timeout)
 9 {
10   CHECK_STATUS_CODE(rtems_message_queue_receive(lx_to_rtems_id,
11                        buffer, size,
12                      opt_set,
13                      timeout));
14
15   irq_insert(TT_ISC_LX_TO_RTEMS, ASYN_EVENT);
16
17   return RTEMS_SUCCESSFUL;
18 }
```

Listing A.11: Function xluna receive from linux

The contracts included in this function are explained as follows:

- Line 1: A precondition that requires that the variable *opt set* must be the global variable *RTEMS WAIT* or *RTEMS NO WAIT*.
- Line 2: Precondition that says that the size of the message must be equal or bigger that 0.
- Line 3: This precondition requires that the variable *timeout* must be bigger than 0 or equal to the global variable *RTEMS NO TIMEOUT*.
- Line 4: A postcondition that ensures that the output of the function will be the global variable *RTEMS SUCCESSFUL*.

The VCGen created 9 proof obligations for this function.

A.1.12 Function isc init

Description: This function is used to initialize the ISC manager. Creates/Initializes write and read resources.

Function C Code:

```
/*@ requires ISC_N_SLOTS==20 && N_FULL_ATT==0 && N_FULL_PRIO==0;
   @ requires N_EMPTY_ATT==0 && RTEMS_TO_LX_ATT==0 && LX_TO_RTEMS_ATT
        ==2;
   @ requires ISC_MAX_MSG_SIZE==512 && N_EMPTY_PRIO==0;
   @ requires rtems_to_lx_id>=0 && n_full_id>=0 && lx_to_rtems_id>=0;
   @*/
rtems_status_code isc_init(void)
{
 8   /* Create write resources */
 9   CHECK_STATUS_CODE(rtems_semaphore_create(N_EMPTY_NAME,ISC_N_SLOTS,
         N_EMPTY_ATT, N_EMPTY_PRIO,&n_empty_id));
10   CHECK_STATUS_CODE(rtems_message_queue_create(RTEMS_TO_LX_NAME,
11                       ISC_N_SLOTS,
12                       ISC_MAX_MSG_SIZE,
13                       RTEMS_TO_LX_ATT,
14                       &rtems_to_lx_id));
15   /* Create read resources */
16   CHECK_STATUS_CODE(rtems_semaphore_create(N_FULL_NAME, 0,
17                 N_FULL_ATT,
18                 N_FULL_PRIO,
19                 &n_full_id));
20   return rtems_message_queue_create(LX_TO_RTEMS_NAME, ISC_N_SLOTS,
21               ISC_MAX_MSG_SIZE, LX_TO_RTEMS_ATT,
22               &lx_to_rtems_id);
23 }
```

Listing A.12: Function isc init

The contracts included in this function are explained as follows:

- Line 1, 2, 3 and 4: All these preconditions it's to guarantee that when entering the function all the properties defined by the xLuna are respected. For example, the value *ISC MAX MSG SIZE* must be 512 by definition.

The VCGen created 14 proof obligations for this function.

References

[1] Bertrand Meyer. Applying "design by contract". 1992.

[2] Reinhold Plosch. Tool support for design by contract. 1998.

[3] Jakob Zwirchmayr. Eiffel â design by contract. 2007.

[4] Dexter Kozen. On hoare logic, kleene algebra, and types. *Department of Computer Science of Cornell University*, 2000.

[5] Nadeem Abdul Hamid and Zhong Shao. Interfacing hoare logic and type systems for foundational proof-carrying code.

[6] Yann Regis-Gianas and Francois Pottier. A hoare logic for call-by-value functional programs.

[7] Common criteria for information technology security evaluation. 2006.

[8] SPEC#. Website. `http://research.microsoft.com/en-us/projects/specsharp/`.

[9] JML. Website. `http://www.cs.ucf.edu/~leavens/JML/`.

[10] VCC. Website. `http://research.microsoft.com/jump/77513`.

[11] Eiffel. Website. `http://www.eiffel.com/`.

[12] Ruby-contract. Website. `http://rubyforge.org/projects/ruby-contract/`.

[13] Frama-C. Website. `http://frama-c.com/`.

[14] William Pugh Jeffrey S. Foster, Michael W. Hicks. Improving software quality with static analysis.

[15] Flemming Nielson, Hanne R. Nielson, and Chris Hankin. *Principles of Program Analysis*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1999.

[16] Jean-Christophe Filliatre and Claude Marché. The why/krakatoa/caduceus platform for deductive program verification. 2007.

[17] Simplify. Website. `http://secure.ucd.ie/products/opensource/Simplify/`.

[18] Alt-Ergo. Website. `http://ergo.lri.fr/`.

[19] Z3. Website. `http://research.microsoft.com/en-us/um/redmond/projects/z3/`.

[20] Yices. Website. `http://yices.csl.sri.com/`.

[21] CVC3. Website. `http://cs.nyu.edu/acsys/cvc3/`.

[22] Kerstin Hartig Juan Soto Christoph Weber Jochen Burghardt, Jens Gerlach. Acsl by example: Towards a verified c standard library. 2010.

[23] Jean-Christophe Filliatre. Why: a multi-language multi-prover verification condition generator.

[24] Coq. Website. `http://coq.inria.fr/`.

[25] PVS. Website. `http://pvs.csl.sri.com/`.

[26] Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, Thomas Sewell, Harvey Tuch, and Simon Winwood. seL4: Formal verification of an OS kernel. In *Proc. 22nd ACM Symposium on Operating Systems Principles (SOSP)*, pages 207–220. ACM, 2009.

[27] Harvey Tuch, Gerwin Klein, and Gernot Heiser. Os verification — now! In Margo Seltzer, editor, *Proc. 10th Workshop on Hot Topics in Operating Systems (HotOS X)*, 2005. to appear.

[28] Matthias Daum, Jan Dörrenbächer, and Sebastian 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.

[29] Gerwin Klein. Operating system verification — an overview. 34(1):27–69, February 2009.

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

[31] University of California at Los Angeles. Website. `http://www.ucla.edu/`, 2009.

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

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

[34] Pascal. Website. `http://www.pascal-central.com/ppl/`.

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

[36] R. S. Boyer and M. Kaufmann. The boyer-moore theorem prover and its interactive enhancement. *Computers Mathematics with Applications*, 1995.

[37] Rudolf Eigenmann and David J. Lilja. Von neumann computers. 1998.

[38] ACL2. Website. `http://userweb.cs.utexas.edu/users/moore/acl2/`, 2009.

[39] Matt Kaufmann. An instructive example for beginning users of the boyer-moore theorem prover. 1990.

[40] David M. Russinoff. An experiment with the boyer-moore theorem prover: A proof of wilson's theorem. 198.

[41] Michael Hohmuth and Hendrik Tews. The vfiasco approach for a verified operating system. *In 2nd ECOOP Workshop on Program Languages and Operating Systems*, 2005.

[42] Jochen Liedtke. On micro-kernel construction. *In Proceedings of the 15th ACM Symposium on Operating systems principles*, 1995.

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

[44] E.G.H. Schierboom. Verification of fiasco ipc implementation. *Master thesis*, 2007.

[45] The Verisoft Consortium. Website. `http://www.verisoft.de`, 2008.

[46] Juergen Dingel and Hongzhi Liang. Automating comprehensive safety analysis of concurrent programs using verisoft and txl. 2004.

[47] Isabelle proof assistant. Website. `http://isabelle.in.tum.de`, 2009.

[48] Mark Hillebrand and Sergey 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.

[49] Sebastian Bogan. *Formal Specification of a Simple Operating System*. PhD thesis, Saarland University, Computer Science Department, August 2008.

[50] Ford Aerospace Communications Corporation. Kernelized secure operating system (ksos). *Executive summary for Defense Supply Service-The Pentagon*, 1979.

[51] 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.

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

[53] Haskell programming language. Website. `http://www.haskell.org/`, 2009.

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

[55] Jochen Liedtke, Uwe Dannowski, Kevin Elphinstone, Gerd Liefländer, Espen Skoglund, Volkmar Uhlig, Christian Ceelen, Andreas Haeberlen, and Marcus Völp. The l4ka vision, April 2001.

[56] John Rushby. The design and verification of secure systems. in eighth acm symposium on operating system principles. 1981.

[57] Raul Barbosa and Johan Karlsson. Experiences from verifying a partitioning kernel using fault injection. *Department of Computer Science and Engineering*, 2009.

[58] Bernardo Patrao, Xudong Guan, and Gustavo Soares. xluna software architecture specification. November 2006.

[59] RTEMS. Website. `http://www.rtems.com/`.

[60] Dexter Kozen. On hoare logic and kleene algebra with tests. 2000.

[61] William R. Bevier. A verified operating system kernel. *Report 11, Computational Logic Inc., Austin, Texas,*, 1987.

[62] Michael Hohmuth and Hermann Haertig. Pragmatic nonblocking synchronization for real-time systems. *In Proceedings of the 2001 Usenix Annual Technical Conference*, 2001.

[63] Hendrik Tews. Microhypervisor verfication: possible approaches and relevant properties. *In Proceedings of the NLUUG Voorjaarsconferentie 2007*, 2007.

[64] 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.

[65] Gregory Duval and Jacques Julliand. Modeling and verification of the rubis microkernel with spin. *In Proceedings of the First SPIN Workshop*, 1995.

[66] Thierry 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.

[67] David Greve and Matthew Wilding. A separation kernel formal security policy. *Rockwell Collins, Inc. Advanced Technology Center*, 2000.

[68] 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.

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

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

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

[72] Tom In der Rieden and Alexandra 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.

[73] Sven Beyer, Christian Jacobi, Daniel Kroening, Dirk Leinenbach, and Wolfgang 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.

[74] Gerd Beuster, Niklas Henrich, and Markus 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.