Investigating Interrupts in the Xtratum Hypervisor using CSP

By

Kevin Hennessy

Supervisor: Dr. Andrew Butterfield

Dissertation

Presented to the

University of Dublin, Trinity College

in fulfillment

of the requirements

for the Degree of

Master of Science in Computer Science

University of Dublin, Trinity College

May 2016

Abstract

My thesis involves investigating interrupts on the Xtratum hypervisor and the Sparc v8 based LEON3 platform. To do this I build a simulation of the system using Communicating Sequential Processes (CSP) with the FDR3 model checker. This sytem contains elements of Xtratum, LEON3 and SPARC V8 and focuses mainly on functionality related to interrupts. I then run a series of experiments to demonstrate how high level properties of my simulation can be checked, and discuss how this could be used to verify properties the real Xtratum C code.

This work is done in the bigger context of a European Space Agency effort to certify the Xtratum LEON3 system for Time and Space Partitioning, a time and memory sharing system for space software.

Declaration

I, Kevin Hennessy, declare that the following dissertation, except where otherwise stated, is entirely my own work; that it has not previously been submitted as an exercise for a degree, either in Trinity College Dublin, or in any other University; and that the library may lend or copy it or any part thereof on request.

Signature:

Date:

Dedication

Dedicated to my amazing family, my parents Margaret and Liam and my sisters Eimear and Orla. Thanks so much for your love and support throughout the years, I couldn't have done this without you.

Acknowledgements

I would like to express my gratitude to the following people who helped me in this research:

- My academic supervisor Dr. Andrew Butterfield for his time and patience in helping me complete this research.
- Artur Gomes for helping me explore and understand Circus and CSP.
- My classmates in Integrated Computer Science
- All the staff and lecturers in the School of Computer Science and Statistics at TCD.

Contents

1 Introduction **6**
- 1.1 Introduction . . . 6
- 1.2 Integrated Modular Avionics . . . 6
- 1.3 ARINC-653 . . . 6
  - 1.3.1 ARINC-653 Architecture Overview . . . 7
- 1.4 High Integrity Software Development challenges . . . 8
  - 1.4.1 Productivity Overhead . . . 8
  - 1.4.2 Cascading Failures . . . 8
- 1.5 Time and Space Partitioning (TSP) . . . 8
  - 1.5.1 Advantages . . . 8
  - 1.5.2 Use Cases . . . 9
- 1.6 TSP using Xtratum and LEON3 . . . 10
  - 1.6.1 What is Xtratum? . . . 10
  - 1.6.2 What is LEON3? . . . 10
  - 1.6.3 Xtratum and Leon as a TSP hypervisor . . . 11
- 1.7 Thesis Evaluation Criteria . . . 11
  - 1.7.1 ESA Requirements Document . . . 11
  - 1.7.2 Research Question . . . 11

2 Background and Related Work **12**
- 2.1 Introduction . . . 12
- 2.2 Examples of Verification Efforts around micro-kernels . . . 12
- 2.3 Process Modelling Language . . . 18
- 2.4 Spark / Ada . . . 19

3 CSP - Communicating Sequential Processes **20**
- 3.1 What is CSP? . . . 20
  - 3.1.1 CSP Overview . . . 20
- 3.2 What is FDR3? . . . 21
  - 3.2.1 FDR3 Features . . . 21
- 3.3 Previous Work with CSP and FDR . . . 22

4 Simulation Design **25**
- 4.1 Introduction . . . 25
- 4.2 System Overview . . . 25
- 4.3 Memory . . . 26
  - 4.3.1 Real System . . . 26
  - 4.3.2 My Memory Model . . . 26
- 4.4 Memory Protection Unit . . . 27
  - 4.4.1 Real System . . . 27
  - 4.4.2 My MMU Model . . . 28

4.5 Interrupt Controller . . . 28
4.5.1 Real System . . . 28
4.6 CPU . . . 29
4.6.1 Real System . . . 29
4.6.2 My CPU Model . . . 29
4.6.3 My Interrupt Controller Model . . . 31
4.6.4 Evaluation . . . 32
4.7 Interrupt Handlers . . . 32
4.7.1 Real System . . . 32
4.7.2 My Model . . . 32
4.8 Peripherals . . . 33
4.8.1 Real System . . . 33
4.8.2 My Peripherals Model . . . 33
4.9 System Integrator . . . 34
4.9.1 Real System . . . 34
4.9.2 My Model . . . 34
4.10 Scheduler . . . 34
4.10.1 Real System . . . 34
4.10.2 My Model . . . 34
4.11 Partitions . . . 35
4.11.1 Real System . . . 35
4.11.2 My Model . . . 35

5 Experimental Procedure **37**
5.1 Introduction . . . 37
5.2 Experiment 1 - Illegal Write Detection . . . 37
5.2.1 Motivation: . . . 37
5.2.2 Design: . . . 37
5.2.3 Assumptions and Limitations: . . . 38
5.2.4 Code Listing: . . . 38
5.2.5 Test Results: . . . 39
5.2.6 Traces: . . . 39
5.2.7 Analysis . . . 42
5.2.8 Conclusion . . . 42
5.3 Experiment 2 - User / Supervisor Detection . . . 43
5.3.1 Motivation . . . 43
5.3.2 Design . . . 43
5.3.3 Assumptions and Limitations . . . 43
5.3.4 Code Listing . . . 43
5.3.5 Test Results . . . 44
5.3.6 Analysis . . . 44
5.3.7 Conclusion . . . 45
5.4 Experiment 3 - User / Supervisor Detection - Expanding the State Space 46
5.4.1 Motivation . . . 46
5.4.2 Design . . . 46
5.4.3 Assumptions and Limitations . . . 46
5.4.4 Code Listing: . . . 46
5.4.5 Test Results . . . 47
5.4.6 Analysis . . . 47
5.4.7 Conclusion . . . 48
5.5 Experiment 4 - Investigating Partition Memory Protection . . . 49

5.5.1 Motivation 49
5.5.2 Design 49
5.5.3 Assumptions and Limitations 49
5.5.4 Code Listing: 49
5.5.5 Results 50
5.5.6 Analysis 50
5.5.7 Conclusion 51

6 Conclusion **52**

7 Further Work **53**

A Appendix A - ESA Interrupt Requirements **54**

B Appendix B - Simulator Code Listing **57**
B.1 Simulator Code 57

Chapter 1

Introduction

1.1 Introduction

My thesis is about investigating interrupts in a hypervisor called Xtratum. (This is an introduction to the whole thesis)

1.2 Integrated Modular Avionics

In [14] from 2015 Integrated Modular Avionics is introduced. IMA is a concept which replaces many seperate processors and line replacable units (LRUs) with fewer more central processing units which applications can share. IMA has led to significant maintainance cost saving and mass reduction in avionics systems since its introduction as a standard in 1987 by the US air force. IMA was first used in the Boeing 777 and has been widely apopted since. Since then IMA has matured and is now used on almost every new airplane entering service today.

IMA hardware usually consists of a group of Core Processor Modules (CPMs) laid out using a standard called ARINC-651 which defines how generic hardware mixed with custom rack based computing hardware should be laid out. This allows for multiple generations of CPMs to be deployed together which is usual for miltary applications which typically use Rugged Off The Shelf (ROTS) CPMs based on OpenVPX or VME-Bus hardware. The paper explains that for military applications IMA is not typically used for flight control systems as there are concerns about how complex redundancy management schemes becomes, as well as the need for these systems to have extremely tight channel synchronization and In this case a federated solution is typically more appropriate, however IMA is used in many other subsystems in miltary, space, and civilian avionics.

1.3 ARINC-653

The ARINC-653 software standard is a key enabler of Integrated Modular Avionics. In [31] from 2008 a standard called ARINC 653 is explored in a paper called 'ARINC 653 ROLE IN INTEGRATED MODULAR AVIONICS'. The paper notes that while the industry has been reluctant to wholly standardize on hardware for IMA, it has embraced software standards, the most prominent being ARINC-653. ARINC-653 is the only standarization effort that takes into account the follwing three key needs:

- Safety critical - As defined in FAR 25.1309 standard, this prescribes highly integrated systems with a Fail **Safe** design concept which aims to assume failures will occur and limit their impact. [36]. According to this standard major failure conditions must be 'remote', hazardous failures must be 'extremely remote' and catastrophic failure conditions must be 'extremely improbable.'
- Real-time - All responses must happen within a prescribed time period.
- Deterministic - All behavior is repeatable and behavior is predictable each time.

ARINC 653 specifies a standardized Real Time Operating System (RTOS) interface for avionics software development. The paper gives three key advantages of this:

- Portability - Avionics applications can be made portable across any compliant ARINC-653 system.
- Cost effective upgrades - By keeping the layer between hardware and application software standardized, the underlying hardware platform can evolve independently of the application software. This ensures there is always a cost-effective upgrade path thoughout the entire life of the aircraft.
- RTOS / Application independence - ARINC-653 establishes a clear boundary in interface between the application and the RTOS, which allows for the independence of the avionics software application and enables concurrent development of the application and the RTOS.

1.3.1 ARINC-653 Architecture Overview

ARINC-653 was natively designed for use with IMA, but can be used with other systems such as federated systems. The following is an overview of the specified software architecure:

- Modules - Avionics systems such as Flight Management and Data Link Communications Management can be implemented in software using generic IMA hardware resources. When targeting ARINC-653, these software systems are packaged in modules called Partition with specific functionality.
- Partitions - A partition is similar to a multi-tsasking application within a typical computer. Partitions have one or more concurrently executing processes and share access to the processor resources. Communication between partitions is possible and it is independent of the hardware requirements of the source and destination partition to allow for hardware from a wide range of suppliers.
- Real-Time Operating System - The RTOS allocates memory region and processor time to each partition.
- Hardware Interface System - Responsible for managing the physical hardware resources on behalf of the RTOS. The boundary between the RTOS and the hardware is specified in an interface called APEX (Application/Executive Interface.) This allows for application portability between ARINC-653 compliant platforms.
- Health Monitor - Manages error recovery of partitions, usually through reconfiguration and restarting of partitions.

1.4 High Integrity Software Development challenges

1.4.1 Productivity Overhead

[27] from 1996 is an analysis of programmer productivity in the European Space agency which examines a a database with data pertaining to 99 software projects in 37 companies in 8 EU member states. The paper aims to compare 'lines of code' to 'process productivity' as metrics for comparing productivity. Lines of code is generally considered to be a bad productivity metric because:

- there is no standard amount of effort that a 'line of code' measures that takes into account all procedural programming languages
- lines of code can be produced by code generators, which make the effort irrelevent
- As the programming language level gets higher, lines of code deceases, making high level languages like C and Ada appear less productive than low level languages like assembly.

Most interestingly for my thesis this study found that productivity varies across ESA companies, and low productivity is often found in companies with high reliability requirements, with high hardware constraints in time and storage. As these are 'fixed costs' when designing software for space the use of modern tools and programming practices are controllable factors that can be used to boost programmer productivity.

1.4.2 Cascading Failures

[21] from 1986 is a NASA study called 'Software safety: why, what, and how' which discusses the safety issues with designing high integrity software. The paper notes that modularization can be used as a design strategy to seperate critical and non critical functionality and ensure that safety critical functions cannot be impededed by the failure of non-critical modules. The goal of this is to change as many potentially critical failures into non-critical failures by reducing the amount of software in the system that affects safety.

1.5 Time and Space Partitioning (TSP)

ESA is attempting to take the IMA concept to the space domain. Time and Space Partitioning is a space-focused implementation of the ARINC-653 standard.

1.5.1 Advantages

[37] from 2009 in a paper called "Time and Space Partitioning in Spacecraft Avionics" by James Windsor, Kjeld Hjortnaes of the European Space Agency which explores the advantages of ESA using TSP.

- Hardware Resource saving - The key drivers for adopting IMA in the aeronautics industry were reduced mass, power, and volume savings for aircraft components. Windsor notes that this mass, power and volume are already a primary concern in the space industry so the benefit for this is limited. However the key problem in the space industry is that software systems tend to get so complex that the only

way to effectively manage them is to split them up, ie decentralize them, meaning hardware is not used efficiently due to redundancy. TSP allows for this logical decentralization without having to add redundant hardware.

- Reduced Integration Effort - TSP can be used to reduce the integration effort of software to the target platform by abstracting functionality at the point of interaction. This means a supplier targetting a TSP system such as an ESA contractor only need to focus on the Verification and Validation efforts that are relevent to them.

- Fault Containment - TSP provides an system where uncorrelated software runs along side each other without interfering in the result of an error. To achieve this a layering strategy is used to contain errors:

    1. Data Level - All memory address spaces of a partition are isolated from the other partitions.
    2. Process Level - If an initializing process fails it will result in actions by the kernel which are limited only o the partition which has the error.
    3. Partition Level - Partitions which overrun their timeslot are controlled by timer interrupts which are managed by the kernel. In the event of an overrun the kernel will forcibly take control away and give it to the next partition.
    4. System Level - In the event a processor suffers from a runaway execution in the kernel, a lockup or untrapped halt instruction etc a watchdog timer interrupt managed by the kernel is used to bring the system back to a safe state.

- Software Criticality - In a non TSP system, a failure with one application can easily propagate to another application, necessitating that all applications be categorized with the same criticality level, even when it isn't appropriate. High criticality systems must be tested with much more rigour than low criticality systems, so this wastes developer time running unnecessary regression tests.

1.5.2 Use Cases

The Windsor paper mentioned in the previous section [37] also explores the main uses cases of TSP for developing software for space:

- Payload Software - The purpose of payload software is to control, command and monitor the mission payload. It is made up of components of differing criticality, for example command and control functionality has a much higher criticality than software designed to process payload data. As these components must co-exist TSP is an appropriate choice to use for making the system easy to develop. On a space mission payload mass is also typically at a premium so TSP is ideal for maximising hardware usage.

- Organizing software teams - TSP allows software developers working on a space project to be split up into several independent but parallel teams who are each allocated a partition who can follow their own software development process without needing to integrate and test their component with other teams. This eases management of the teams and decreases delays and cost overruns.

- System maintainance - Software that is running on an actual spacecraft is typically difficult to upgrade and patch as extensive verification is required before the change

can be made. TSP allows for the starting, stopping, upload and removal of hosted applications with minimal impact on the overall system. For example this means that operators of commercial sattelite running TSP would be able to safely upload new payload software to the live satellite without requiring ESA or the satellite manufacturor to be involved.

- Security - TSP allows for a high security partition to be run nd be protected from other malicious applications on the same system. An example of this is being able to safely run a military application on a commercial satellite without data from it being breached by another partition.

1.6 TSP using Xtratum and LEON3

1.6.1 What is Xtratum?

Xtratum is a hypervisor developed in 2004 by the Real-Time System Group in the University of Valencia. Xtratum is what is known as a type-2 hypervisor, as it runs on the bare metal of the hardware, which is different to a type-1 hypervisor like virtualbox, which requires an Operating System to run.

Xtratum was designed specifically for real time safety critical software, and includes all functionality required to build safety critical systems based on ARINC-653, such as scheduling policy, partition management, inter-partition communications, health monitoring, logbooks, traces etc. [23] Additionally, xtratum has been specifically designed to efficiently meet real-time constraints: [25]

- Static data structures - All data structures used by Xtratum are pre-defined at build time based on a configuration file. This is done so that the exact resource overhead used by Xtratum is constant.
- Non Pre-emptive code - This is a desirable feature in most real time operating systems but there is no benefit in Xtratums case as it just adds complexity. The code is faster and requires no fine grained critical control sections.
- Deterministic Hypercalls - Xtratum provides only the bare minimum functionality needed to guarantee the spatial and temporal isolation of partitions
- Interrupt Occurance Insolation - When a partition is executing, inter-partition interference through hardware interrupts are minimized by only enabling interrupts managed by the partition.

1.6.2 What is LEON3?

LEON3 is a 32 bit RISC processor architecture based on the SPARC v8 architecture which is ideal for simple sytem-on-a-chip (SOC) designs. [1] [25] It was initially developed at ESTEC in the European Space Agency in 1997 as an open, non-proprietary architecture design that would be capable of meeting future performance requirements, software compatability, and low system cost.[22] Since 2008 the Leon Architecture has been maintained by Aeroflex Gaisler. It is provided as a sythesizable VHDL model and available under the GNU GPL licence. Leon3 is designed to be highly configurable with 1-16 processors and optional modules such as IEEE-754 floating point unit, SPARC reference Memory Managment Unit, and Interrupt Controller unit. [24]

All processors in the LEON series are based on the SPARC-V8 RISC architecture. [13] SPARC v8 processor designed for simplicity and high performance. On May 1st 2015 Leon3 was certified as being SPARC conformant.

1.6.3 Xtratum and Leon as a TSP hypervisor

[26] is a paper from 2010 called "XtratuM for LEON3: an Open Source Hypervisor for High Integrity Systems" which discusses using Xtratum and Leon3 as a high integrity hypervisor. Leon3 is especially interesting to ESA because as the paper notes it is the first Leon Processor that is able to perform virtualization with full spatial partitioning due to being the first Leon processor to have a Memory Management Unit. Xtratum can be run on Leon2 which proves high integrity virtualization is possible on a SPARC v8 architecture but the lack of a Memory Management Unit on Leon2 makes it impossible to implement full spatial partitioning, which is necessary for TSP.

ESA is interested in using Xtratum for TSP and has been investigating verification efforts of Xtratum [33].

1.7 Thesis Evaluation Criteria

1.7.1 ESA Requirements Document

As a part of my thesis I was given a set of ESA functional requirements that describe a Time and Space Partitioning system. [11] There are 176 of these requirements. These 176 requirements are categorized as either single-processor or multi-processor. Requirements that are considered to be integral to the system are designated as CORE and other requirement are designated as EXTENDED. For my thesis I am focusing on the single-processor CORE requirements only. These requirements describe functionality related to Interrupts, Memory Management, Partition Scheduling, Timer Management, Inter-Partition Communication, Health Monitoring, and system integration.

Note: See Appendix A for a listing of requirements that are relevent to my thesis.

1.7.2 Research Question

ESA desires to link the Xtratum C code to a formal proof of its behavior, and show that proof meets the requirements that they have specified. My thesis is about assisting that process by building a model of the system that captures the behaviour of how interrupt behave in the system. Interrupts are a pain point in the debugging of hardware and as such ESA is investigating formal modelling techniques to verify and validate this area. My thesis is about assisting this process by building a model of Xtratum that can be used as an intermediate verification layer between Xtratum running on Leon3 and a future formal proof of behavior.

Chapter 2

Background and Related Work

2.1 Introduction

I did a lot of reading about kernel verification in an attempt to find a strategy to investigate Xtratum. In this section I will explore some of the approaches that I considered.

2.2 Examples of Verification Efforts around micro-kernels

There are many examples of attempting to use Formal verification to prove properties of high integrity systems. In this section I will explore similar efforts.

Because of their small size there have been several examples of formally verifying micro-kernels. One of the most well known examples from 2009 is the seL4 Kernel Verification effort [17], where an extremely simple micro-kernel was developed with the help of formal verification and comprehensively verified to be correct. This process was done by creating a chain of formal proofs from high level safety requirements all the down to the executable machine code.

This means that assuming the specification is correct , the kernel functions entirely as intended. The kernel was designed for real world use, and does not trade the average case execution time for formal verification. The paper acknowledges that worst case execution time may be affected, but this is difficult to check in the absense of other similar efforts.

sel4 was not designed with a 'bottom up' approach which would be the conventional way to design an OS kernel to make efficient use of the hardware. seL4 was not designed with a 'top down' approach which would be the conventional way to design a program using formal methods. sel4 used an approach that picked the best of both of these perspectives. A verifiable subset of the programming language Haskell was used to design a prototype of the Kernel which is derived from the informally stated, english language requirements. This subset does not include complex langauge features such as laziness, with all functions proved to terminate and not using type-classes extensively and thus is verifiable. This prototype can be translated into the Isabelle/Hol theorem prover to form a design level specification of the Kernel. This model can be used to run hardware simulations which allows developers to run and test low level code from the user and kernel perspective. After many iterations when the specification stabilized, this Haskell prototype was then manually translated into a verifiable subset of C, in the C99 standard. This C code is the final Kernel implementation. To manipulate hardware directly, the programmer has to occasionally go outside the semantics of C and

use assembly language. The paper does not model the effects of certain direct hardware instructions which are relevent to the formal correctness such as cache flushes and Translation Lookaside Buffer flushes because they are outside the abstraction layer of the program semantics and these are checked with traditional testing. (While the kernel is actually executing.) The paper notes that more detail could be added to the machine model to simulate this.

seL4 obtained a number of 'firsts' during its development. It was the first proof of functional correctness of an OS kernel. It was the first proof of correct translation from C down to binary. It was the first to prove security properties like integrity and intransitive noninterference in an OS kernel. For an OS supporting full virtual memory It was the first analysis of worst case execution times.

In [12] from 2013 a formal verification of information flow in the PROSPER seperation kernel for ARMv7 is demonstrated. Information flow security is ensuring that software in different security domains such as inside different partitions are sufficiently isolated from each other. Any communication that takes place between partitions must be authorized. It must appear to each component system that they are executing on a seperate, isolated machine. The paper focuses on running an "untrusted" component such as a smartphone software stack that interacts with a set of trusted services such as a virtual sim card. The minimum required functionality for the Kernel is isolated component resources, a communication system that mimics communication lines in a physicaly distributed system, and a sheduling mechanism for the shared resources.

This paper notes that significant progress has been made in this area thanks to seL4, Microsoft VCC, and Green Hills' CC certified INTEGRITY-178B separation kernel.

The goal of the papers approach is to use a top level specification (TLS) to make communication between partitions explicit and information flow is analyzed in the presence of such an intended communication channel. The TLS will directly formalize the set of computational paths allowed and required at the implementation level. This means that there should be no way for partitions to read and write to each others memory, registers etc unless through an allowed channel by both partitions in collaboration. The paper aims to carry through the specification, implementation and correctness proofs all the way from TLS to an implementation in ARMv7 which is proved correct at the instruction reference sematics level. It will do this using the Hol4 model of ARM developed at cambridge. (can reference if wanted.)

For the verification to take place models are built. An "ideal model" is built which is an exact formulation of the TLS which is compared against a "real model" which is obtained by building on top of the Cambridge ARM HOL4 model, extended with a simple MMU. A tool ARM-Prover (developed in HOL4) is used to verify that a partition does not perform any isolation-breaking operations while executing and the processor state switches correctly on transition from user to supervisor mode. The proofs involves traversing the full ARM instruction sets and thus are costly. The Hol4 code was then analzed by BAP, the binary analysis program by translating the code from Hol4 to BIL.

The paper acknowledges two main limitations to their approach.

Their model counts instruction cycles instead of real clock cycles, but in their implementation real clock cycles are used. They say that it is non-trivial to extend their

model to a more realistic representation of time as phenomena like cache delays and instruction pipelining come into account which are outside the modelling scope of the work.

Their model has difficulty modelling unpredictable states. When executing unpredictable behaviour ARM does not allow the execution of instructions to perform any function that are above the current priviledge level. To get around this the model introduces error states outside the Lemma's they are proving. The paper acknowledges that this is not satisfactory as partitions can exit the scope of the model by entering an unpredictable state.

In [20] from 2009 the verification of the Microsoft Hypervisor 'Hyper-V' with the formal verification suite for concurrent, low level C code 'VCC' is demonstrated. 'VCC' is being actively developed by researchers based on the verification of 'Hyper-V.' The paper acknowledges that Formal Verification of a hypervisor is difficult as it was not written with verification in mind and identifies several key challenges of creating a verification tool for it.

Challenge 1 is that verification has to be done at the code level with assertions rather than a high level interactive theorem prover in a typical industrial process where a lot of developers and testers are developing the code like theirs.

Challenge 2 is that the hypervisor is being developed in the C programming language which is difficult to verify memory in. It means that memory safety has to be explicity verified because of C's weak, easily bypassable type system and use of explicit deallocation instead of garbage collection.

Challenge 3 is that hypervisors make heavy use of lock-free syncronization. The paper explicity mentions address translations as a difficult as they are gathered asynchronously and non-atomically, requiring multiple reads and writes to the page table.

Challenge 4 is that it is difficult to keep annotations very tight to th ecode as there is no native way to show that concurrent data types simulate some simpler type, a typical way to prive properties about that concurrent data type.

Challenge 5 is that the Hypervisor is written in C mixed with assembly. Any verification scheme will have to take into account any subtle interations between the two and how they relate to the hardware resources.

The paper acknowledges that usually the correctness of an implementation is veridied by proving a simulation between a simulation and an abstract model. (This validates our approach with CSP.)

The VCC tool is use to verify the low level concurrent C code of the hypervisor. The annotations will be integrated into the codebase as comments and maintained by the developers. VCC performs static analysis of functions using invariants of types and the contract of function calls before and after execution. The Invariants guarantee things like that objects of the same type with different addresses don't overlap and thus behave like objects in a modern type-safe object oriented language. In VCC Objects may occupy the same space as pointers to C structs, but also with sub-structures and arrays, similar to the language spec sharp.

For verifying the hypervisor, they use a model of the x64 processor architecture, described in C. "Ghost" functions are used to operate on a "ghost" data structure that represents the processors state. This model is used to verify the low level correctness of the hypervisor at the level of assembly code and at the top level specification to verify

the hypervisor correctly simulates x86 machines for the guests. The goal of the proof is to enforce a coupling invariant between the volatile ghost state of the top level model and the implementation of Hyper-V.

This verification effort is still ongoing with no published updates since 2010. [10]

In [32] from 2010 the formal analysis of the INTEGRITY-178B kernel is demonstrated. The purpose of this effort was to be the first to get a Common Criteria for Information Technology Security Evaluation level EAL6+ certification for a commercial real time operating system kernel. In USA, the US National Information Assurance Partnership (NIAP) performs Common Criteria evaluations and EAL7 is the most stringent level.

The Kernel implements ARINC-653. A security benefit of this is that applications cannot signal another application that it is not authorized to, be it maliciously or otherwise. The Federal Aviation authority in the US enforce a safety guideline called DO-178B which has 5 criticality levels A-E, A being the most critical which describe the impact of aircraft safety incase there is a failure. INTEGRITY-178B is designed to allow mixed criticality applications to run and prevents faults from cascading to other applications by ensuring other applications do not miss their 'deadlines' with a fixed timeslot for each.

For a Common Criteria analysis, there are three levels of rigor to look at the system at.

(1) Formal - Where machine verified, mathematical proofs are required. For INTEGRITY, this includes the Security Policy model, the Functional Specification.

(2) Informal - The properties such as security are justified in natural language, but to a very high detail.

(3) Semiformal - a mix of both. For INTEGRITY, this includes the High Level Design, Low Level Design, Representative Correspondance between entities.

To model the system state, the paper uses the ACL2 theorem prover, based on a functional language. Existing formal specifications of separation properties such as GWV secuirty policy were not expressive enough to state anything meaningful about INTEGRITY-178B so a new policy GWVr2 was developed. To do this the system has to be modelled as a graph of state transitions. Data is ordered like Unix paths to allow for nested data structures. Operators are also defined to update and query the state.

The Common Criteria forbids the low level design specification and the source code to be one and the same. Establishing correspondence between them is a difficult, time consuming procedure. The main goal of the modelling procedure is to create a compelling argument that the behaviour of the system is captured accurately.

When using a functional language for modelling the paper identifies several challenges.

(1) - ACL2 requires proof of termination for functions so reflexive recursion cannot be modelled. Reflexive Recursion is where two succesive recursive calls are made, the latter taking an argument something computed in the first. To get around this the

recursion must be unrolled.

(2) - It is difficult to make the model an extremely close 1:1 corresponance to the source code as the reserchers intended. loop constructs and recursion are handled differently in functional languages. Global variables in C must be passed as state. Functions in ACL2 must return a state, whereas with C they do not have to return anything.

(3) - Asynchonous interractions with the world outside the Kernel such as interrupts are difficult to model.

After modelling this and verifying the various layers, in 2008 Common Criteria EAL6+ was awardedin September 2008.

An example of verifying a processor in high detail can be seen in [6] from 2003 where the formal specification of the VAMP processor is verified and the processor implemented on a FPGA (Field Programmable Gate Array) chip. VAMP runs the DLX RISC processor architecture so is roughly similar in complexity to LEON.

This model featured very high fidelity models of the main features of the processor, a full DLX instruction set, FPU, instruction/data cache, delayed branch, Tomasulo scheduler and maskable nested precise interrupts, similar to what is in LEON. The verification of all these seperate components took 8 person years, 2548 Lemmas and 88622 proof steps to formally verify in the theorem proving tool PVS. These models were so complex that an additional person year had to be taken to 'patch' the links between these seperate models together as despite their planning the interfaces between the parts did not exactly match up. A model of this detail is far beyond the scope of my project.

An example of having formally verified software that can be built upon is descibed in [18] from 2005 where the verification and implementation of a compiler is discussed. This Compiler is part of the Verisoft project which aims at the formal verification of entire computer systems. C0 is the main language of the Verisoft project and is the language the compiler consumes. The target arhcitecture for the Verisoft C0 Compiler is the DLX assembly language of the VAMP verified processor.

This compiler is to be used to compile a subset of the C programming language called C0. This verification effort is notable for my project as it aims towards thorough verification of the entire system including hardware, system software and the compiler itself. This is similar to the problem that I face with Xtratum.

Compiler verification is a well explored field but Verisoft is different because of its complex requirements. As they must do a pervasive verification approach, just verifying the compiliation is not sufficient. A verified implementation of the compiler is required to ensure properties proven for the source code level applications also hold for the compiled code. Another difficulty is that Interleaving of other 'user' programs on the hardware makes it difficult to formally model it using 'large scale semantics' where the overall results of the simulation are described. As most of the verification for the project is done using Hoare Logics, extra work must be done to 'import' the terminating sections of code described into 'small scale semantics', the type of formal language where individual steps of a computation are described. The large size of the program makes this even more difficult

More difficulty comes from the target architecture. Another difficulty is for verify-

ing dynamic data structures, the representation of memory for the destination machine DLX is different to the memory representation of the compiler. Data must be alligned in DLX. The C language semantics do not cover this. They state that dynamic memory handling should be handled by the standard libraries. For branching, DLX also makes use of delay slots which complicates proofs as instructions might appear to be in the incorrect order.

The paper uses Isabelle/Hol to capture the compilation of the C0 code to VAMP machine code and prove the implementation of the Verisoft C0 compiler is the same as the specification. As of 2008 The compiler has been verified in the context of pervasive system verification. [19]

In [5] from 2009 the formal verification of the functional correctness of the pikeOS micro-kernel is explored. PikeOS is an embedded type-1 partitioning hypervisor written in C and very similar to Xtratum. This research is being carried out as part of the Verisoft XT Avionics sub-project. This work is built upon work done for Verisoft's C0 compiler from 2005 [18] in the Verisoft I project, of which the Verisoft XT project is a successor.

The VCC formal verification suite from microsoft research [20] is the tool that they use. The way the verification works is the annotated C code of the kernel is compiled with VCC into BoogiePL, an impertive language that includes properties of the C program in assertion form. Axiomatic descriptions of certain parts of the C programming language are captured seperately in BoogiePL. These are then translated to first order predicate logic formulas and are passed into the theorem prover Z3, which is used to verify the original C code of the Kernel matches the specification.

low level function verification

The paper notes that the large amount of assembly code present in PikeOS was a challenge to model as it is commonly used inline using the doubleunderscore asm doubleunderscore keyword and its sometimes contains priviledged instructions for changing mode which are usually not considered in formal definitions of instruction set semantics for programs in user space. The VCC is also not able to interpret machine instructions. Because of these challenges they decide to formalize their own hardware and assembly abstract formal model. (You will see later in this dissertation that this is something that I will do with Xtratum)

In their model assembly code is defined as a transition over hardware elements.

First they must identify the relevent hardware components and make an abstraction. They divide the hardware components into ones that are changed by the C code of the Kernel (eg registers that are visible to the programmer) and those that are not EG eg special purpose and system registers. Caches and TLB's are ignored as in a single processor environment they are invisible to the programmer. This model is implemted as "ghost state" of the system, basically a data structure in the C code which does not modify the structure of the C code and corrupt the verification effort.

Once they have done this they define specification functions for each assembly instruction that is equipped with pre and post conditions that reflect the functionality of that assembly instruction, which update the 'ghost state'. These are then integrated into the PikeOS code using a parser, replacing the assembly, allowing functions with inline assembly instructions to be annotated and automatically verified in VCC. Inter-

rupts are disabled while these functions are executed. Compbined with the hadrware model This allows them to verify functionality across all levels of the kernel

An interesting example of using these annotations to check useful properties is explored. They need to track the value of the stack pointer in a fucntion with inline aseembly. (This is the base address of the process) The problem with this is the stack pointer value is only known by the compiler and does not exist on the C level. They solve this problem by tracking context switches in the hardware model and taking advantage of the knowledge that the stack pointer lies in a range of addresses whose base is determined during context switch when register 1 points to a stack frame of a specific process.By tracking context swicthes in the hardware model They are able to narrow the possible value down from a range to an exact address using this method. This is similar to work I do with interrupts and context switching later in my thesis.

PikeOS is developed to the DO-178B avionics safety standard. This requires all requirements have lots of system and unit test with a high coverage. The explosion of states makes this difficult.

2.3 Process Modelling Language

As part of my thesis I investigated a tool called Process Modelling Language (PML). In this section I will give an overview of some similar work in this area.

In [29] from 2003 an approach for representing knowledge intensive processes with low fidelity models is explored using PML. The paper proposes an approach that captures the two main difficulties:

1) Tasks change continuously in response to knowledge

2) Informal communications are often heavily featured in knowledge work, which is difficult to capture.

A low fidelity model is a model that captures the main activities of a process and the order in which the steps are performed. It does not aim to capture every little detail of the process but describe it in broad strokes. This can be used to formally describe complex activities.

The paper gives the example of describing the software development process by looping through the actions EDIT, COMPILE, TEST, DEBUG. The act of procucing code has many possible transitions between these four actions which is difficult to elegantly specify in a model without having many edges for each case. To describe this in an elegant way with a low fidelity model in PML we can consider it in terms of resources, such as the TEST actions consuming a resource called 'exec' which is produced by the COMPILE action. This results in each actor having several different possible paths they can perform.

The paper describes the process of modelling having the following advantages:

1) Low fidelity models remain relevent throughout the entire software development process as since they only describe high level details they continue to be accurate depictions of the process under consideration.

2) Trust between entities is not needed for actors in the system because they communicate through a shared resource. Because of this new actors can be easily added to the system.

This is a good example of building a simple model from a set of high level requirements like what I am doing with my thesis.

Enactment Theory is the theory of how people act in organizations. This concept is derived from the Actor Model, which is a comouter science model that treats 'actors' as the primitives of computation. Enactment theory is an ideal theory to investigate for my thesis as hardware elements in the xtratum /leon system interact with each other like actors in an organization.

In [30] from 2004 Process Modelling Language is used to define a complex user interaction using low fidelity models as above.

From this an 'Enactment Engine' is created which aim to capture the idea that an expert actor may not need the process to help him or her perform the task, so they can query the model only when necessary, but a novice actor can query the model at every step if needed.

2.4 Spark / Ada

Ada is an imperative, object oriented programming language commonly used in the Avionics Industry. Ada was the first ever object oriented programming language to become an international standard after it was created by the US Department of Defence to consolidate the hundreds of programming languages that they used. Today Ada is widely used in real time, safety critical systems. Over 99.9 percent of the code in a Boeing 777 is Ada code. [3]

For my thesis I am most interested in Spark, which is a verifiable subset of the Ada programming language. Spark has been used to verify numerous safety critical systems in industry for over 20 years. The most interesting in terms of my thesis is the open source project called Tokeneer. Tokeneer is an open source program to model a secure ID station and was written as a case study by the NSA in the United States using Spark as an example to the high-integrity software community of how high security systems should be created. The Tokeneer project is relatively small (around 10,000 lines of code) but has been extensively studied since it was open sourced in 2008 and only two breaches of its specification were found, one by static analysis, one by code review. [39]

While I am not actually designing a project from scratch I have yet to see exactly how useful Spark / Ada will to be my project, it may be useful for building models of the requirements and matching it to Xtratum.

Chapter 3

CSP - Communicating Sequential Processes

3.1 What is CSP?

CSP was first introduced by Tony Hoare, the british computer scientist who is best known for inventing Quicksort and Hoare logic, in his 1978 paper 'Communicating Sequential Processes' [16] I used CSP extensively in my thesis. In this section I will explain how it works.

3.1.1 CSP Overview

In [8] from 2009 Neil Brown of Communicating Haskell Processess gives a thorough overview of CSP.

Events

Events in CSP are atomic entities that represent actions of a process. Events are used to synchronize processes. Events can act like a channel for passing data between two processes or can be used to synchronize processes without sending data.

Processes

CSP processes are a sequential chains of events. The simplest operator in CSP is the sequencing operator '`->`' which combines an event and a process to produce a new process. There are two primitive processes in CSP, STOP which represents deadlock and SKIP which represents a successful termination.

Internal Choice

The deterministic choice operator '`|~|`' is used to allow a process to choose between two behaviors, as defined by the environment. For example, a process that communicates events a and b can either behave like P or Q depending on what is communicated to it by the environment.

```
(a->P)|~|(b->Q)
```

External Choice

The future behaviour of a process is defined as the choice between two processes, however it does not allow the environment any control over which is selected. It is denoted by

the [] operator.

Interleaving

The interleaving of processes is performed by the ||| opeator and used to represent concurrent activity. Events from the processes P and Q in P|||Q are arbitrarily interleaved in time.

Interface Parallel

Concurrent activity that requires synchronization between processes is denoted with the [| {| |} |] operator with all synced events in the middle.

CSP example

The classic example of CSP is a chocolate vending machine with three events, "coin", representing the input of a coin, "card" reprenting the input of a card, and "choc" representing the delivery of chocolate. This system can be encoded like so:

```
1 VendingMachine = coin -> choc -> STOP
2 Person = (coin -> STOP) |~| (card-> STOP)
```

This model has two processes, VendingMachine and Person. These processes can be run in parallel, synchronizing on their common events (ie "coin"), so that their interaction with each other is modelled. Person has a deterministic choice to use "coin" or "card", but since VendingMachine does not have a "card" event that path will never be followed. There is also the notion of non-deterministic choice which will pick one option randomly and which would deadlock in this case.

3.2 What is FDR3?

In [15] from 2014 an overview of the FDR3 tool is given. FDR3 stands for Failure Divergence Refinement and is the most commonly used model checker for CSP. FDR takes a list of processes written in a lazy functional language CSPM.

3.2.1 FDR3 Features

Deadlock testing:

FDR3 can be used for checking Deadlock Freedom. A deadlock is when two processes are preventing each other from accessing a shared resource, halting progress in the program. While being able to detect deadlocks is useful, the real power of this involves defining "checker" processes that synchronize on your main model at certain points and deliberately deadlock when a certain condition occurs, allowing you to build up progressively more complicated tests out of this relatively simple one.

Divergence testing:

FDR3 can be used to test for Divergence freedom. Divergence Freedom is when the program will always terminate in a visible non-exception state. [7] This is a useful test to ensure your program is not doing an endless series of hidden actions.

3.3 Previous Work with CSP and FDR

In [4] from 1995 the specification and verification of the virtual channel processor hardware of the INMOS T9000 Transputer is demonstrated. This is one of the earliest and important examples of using CSP as an industrial model-checking tool.

Development of the T9000 transputer was started by INMOS, a british semiconductor company, in 1989. The T9000 shared elememts of its predeccessor T800 such as instruction set capability, but needed a completely new design for the virtual channel processor hardware. The Virtual Channel Processor (VCP) is a hardware element that allows N number of logical connections between two processors to share a single physical communication link. Automatic verification is required to some degree as the protocols that are required in the hardware implementation are quite complex.

Tasks like this are called VLSI, or Very Large Scale Integration which is the process of creating an integrated circuit by combining a large number of transistors onto a single chip. The paper notes that in the future as transistor density increases designs are better constructed from independent modules.

The T9000 is designed as a series of sub-components. Many of these specications are represented in many different forms such as the English Language specification, Z specification, timing diagram, occam functional simulation, VHDL description and the sillicon layout. The paper notes that being able to model check these against each other is an essential took available to the engineer for a complex system like this. Visual inspection for comparison is not practical as it is error prone and time consuming. Hardware systems are getting more complex all the time and it is incompatible with the complexity. It also makes fast iteration difficult.

Similar to LEON, most of the control logic is produced from synthesis of VHDL descriptions. The paper notes that the design of the 'cells' used by the synthesizer requires extensive digital logic design knowledge (verified by simulation) the design of the input will not require the engineer to know this knowledge. It notes there is a trend towards high level behavioural description programs like CSP paired with an associated specification and proof tools.

The paper uses two model checking tools. An early version of FDR and a custom tool written in SML. They designed their CSP code by creating models that use the alphabet of the process as parameters because it is less error prone as it is similar to the idea of strong typing in programming languages and also produces a less verbose program.

They use FDR to use the standard CSP operators such as run (do anything except diverge or refuse event), chaos (do anything except diverge), parallell composition of processes (syncing on specified events), external choice, internal choice, event hiding, event renaming etc. They also define some custom operators like logical conjunction and logical disjunction of processes which is made possible by choosing operators that when both operands be finite state machines, it will result in a finite state machine. This means that all of their CSP models can be automatically model checked.

From the VCP specification a channel is a point-to-point link between two processes. A processsor may execute any number of processes but can only use one channel at a time

for input and output. Processes that perform operations on a channel are de-scheduled until the value passed into the channel is picked up by the other process. The VCP can be specified as a the interleaving between a number of these independent channels.

The paper uses CSP to check 3 properties of the VCP:

Condition 1 - Safety - A safety condition is for specifiying that nothing undesirable or dangerous can happen. In terms of the T9000 VCP, this means specifications like an input to a channel cannot terminate before it has begun. There are 4 variables needed to describe this, #in #out for the number of inpputs and outputs which have begun and #run(in) #run(out)for the inputs and outputs that have terminated. There is never more than one active input or output. There is a bound of 2 on the combined number of active inputs and outputs. Because each value can take on a finite range of values it is ideal for modelling as a state machine / in CSP. As the VCP performs events, these variables will change.

Condition 2 - Reliance - The reliance condition is used to verify that the process is behaving within its specified operating conditions. In terms of a channel in the VCP, the rely condition means that traces are composed of alterating sequences of in-run(in) and out-run(out) events.

Condition 3 - Liveness - the liveness condition is used to show that something useful must happen. For example in the model of the VCP certain events act like commands (eg: ChannelId.in, ChannelID.out (pseudocode)) and certain events act like reactions to those commands (ChannelID.run(in), ChannelID.run(out)). The VCP can shedule its reactions in any order and can therfore refuse all except one at any one time. Liveness in this case means that the process can never refuse to accept a command.

Three models are built and then linked together, output model, input model, link model.

Due to various performance problems and delays the T9000 transputer project was eventually abandoned in 1997. [38]

CSP has been used in space technology before. In [9] from 1997 CSP is used to verify a fault tolerent data management system for the International Space Station (ISS) written in the occam programming language and created by Daimler-Benz Aerospace. The main motovaion for the verification effort was to ensure that the code that handled process communication was deadlock free. The program under verification is split up into four communicating lanes, each with 3 layers, an application services layer (ASS), a fault management layer (FML) and an avionics interface. (AVI)

The goal of the verification effort is to derive a CSP abstraction which summarizes the important parts of the process specification and the to use FDR to check the abstraction is a deadlock free, then produce a refinement (a translation from a high level spec to a real executable program) using real occam code.

The verification approach uses 3 techniques:

Technique 1 - Model Checking - This involves using the technqiue of refinement which is

Technique 2 - For just analysing deadlock freedom we only have to focus on the aspects of the program that are related to communication, thus we can create a simplified CSP model. The CSP specification A(P) is called a valid abstraction of the process P

if whenever P deadlocks, A(P) does also. This is a challenge to create and the paper uses the following techniques to do it. Remove sequential code that does not influence communication behaviour. Reduce every occam channl protocol to the set of values influenced by the communication behaviour. Every conditional statement (if statement) is replaced by an internal choice operator in CSP (P [] Q) or a CSP "if" statement. This appproach takes advantage of the high correspondance between Occam and CSP which I will not have the luxury of in my project.

Technique 3 - Composability - CSP provides several high level operators that are used to produce new processes from existing ones. These are the parallell operator (||) and the interleaving operator (|||). Refinement is preserved when these operators are composed and thus if the state space is too large to run the entire system in parallell we can create simpler processes which are refinements of the more complicated ones and reduce the state space that way.

The paper concludes with a review of CSP as a verification tool which they see as very positive because the project under scrutiny is of real world size at around 20,000 lines of occam code. The paper note that a failure of conventional formal methods is the inability to scale up to the size of a realistic application. The paper also says that based on similar efforts using conventional formal verification techniques would take approximately 4 times as long. They note that they do not think they would have reached the same level of trustworthiness with conventional tecniques. The verification of the AVI and FML subsystems uncovered several deadlock situations that had not have been found with testing. [34]

Chapter 4

Simulation Design

4.1 Introduction

For my thesis I designed and implemented a simulator for Xtratum running on the Leon3 architecture. Leon3 is an architecture which contains a Sparc v8 CPU. The focus of my simulation was Interrupts and Memory because they are the requirements I was interested in. In this section I will document the simulator and how it differs from the real system.

To create this model I combed through the following documents: the SPARC v8 manual [28], the Leon3 manual [35] and the ESA requirements document. [11] Where there are ambiguities between SPARC and Leon3 documentation, Leon3 takes precedence.

4.2 System Overview

The ESA specification [11] describes a partitioning system as a system made up of a partition kernel and a set of partitions. Each partition provides an isolated environment for the software running on it as all resources assigned to the partition are completely independent from the others. If software requires access to another partitions resources, it must do so using only mechanisms the kernel provides. The partitioning kernel is executed in supervisor mode and virtualizes hardware resources such as peripherals, interrupts, memory and CPUs. The partitioning kernel must provide the following:

- Memory Management - The partition kernel uses hardware elements such as the MMU to ensure spatial isolation.
- Interrupt Management - The partitioning kernel handles all interrupts. They may be forwarded on to partitions depending on their nature. Virtual traps can be provided to partitions.
- Scheduling - Partitions are scheduled on a cyclic basis to enforce temporal isolation. Several different cyclical scheduling plans can be used. The partition kernel maintains a major time frame of fixed duration and partitions are allocated one or more smaller timeslots within this major frame. A configuration file is used to specify the order of partition activation within this major frame. The kernel is responsible for syncing partitions with the scheduler via clock.
- System Integrator - The system integrator is responsible for configuring the entire system by interpreting configuration on boot and perfoming tasks like allocating virtual hardware to partitions and setting up the scheduling plan.

- Tracing facilities - It is desirable for the partition kernel to provide a facility to store and recieve traces generated by the kernel and partitions for assistance in debugging, or further developing the application. (Linking these generated traces with a CSP model might be a useful for checking for formal correctness of the source code.)
- Inter-partition Communication - Partitions can pass messages between each other using known channels controlled by the seperation kernel. This communication system resembles the one defined in the ARINC-653 standard discussed earlier.
- Health Monitoring - The health monitor is used to detect anomalous states that partitions may enter into in an attempt to reduce the possible consequences. The HM aims to capture errors that are outside the scope of the program that is running in a partition. For example, a call to malloc returning null in C due to not enough memory would be handled by the program, but a processor trying to perform an undefined instruction would be an error for the HM.

4.3 Memory

4.3.1 Real System

The memory model in SPARC is called Total Store Ordering. [28] All SPARC versions must provide this as minimum. Leon3 uses this memory model as it is the default. [35] Total Store Ordering is a scheme where a processor can do a read to memory location B before a write to location A is seen by all processors, but it cannot do a read from A until the write to A is seen by all processors. Multiple writes however must occur in program order.

SPARC also allows for a more complicated memory model called Partial Store Ordering. [28] In PSO the processor can re-order writes so that the sequence of stores to memory is not identical to the sequence of stores issued by the CPU which can boot cache performance. [2]

SPARC defines 4 address space identifiers: user instruction, user data, supervisor instruction, and supervisor data. The architecture allows 256 possible address space identifiers but only the above 4 are used by SPARC.

4.3.2 My Memory Model

Description:

My memory model is made up of one process. It has one piece of state which is a CSPM map which maps addresses to data. Reads and write events sync with the CPU and when they happen data is passed across a bus, represented by an event that syncs with the memory management unit, which will block an illegal write at this point. On a sucessful write, the map is updated using the CSPM mapUpdate function via the write evnt. On a sucessful read, data is passed to the CPU via the read event.

CSPM code:

```
1
2 mem_init = (| a0 => d0, a1 => d0, aDC => d0, aX => d0 |)
```

```
datatype Bus_Dir = r | w
channel read, write : Data   --synced with CPU
channel bus : Bus_Dir.Mem_Region --synced with MMU
channel badaccess , mmuOK          --synced with MMU

--m is the memory map

MEM_BUS(m)
=   tick -> ( MEM_BUS_READ(m) [] MEM_BUS_WRITE(m) [] MEM_BUS(m) )

MEM_BUS_READ(m)
= bus.r?a -> ( ( mmuOK -> read!(mapLookup(m,a)) -> MEM_BUS(m) )
              [] (badaccess -> MEM_BUS(m) ) )

MEM_BUS_WRITE(m)
=  bus.w?a -> ( ( mmuOK -> write?d -> MEM_BUS(mapUpdate(m, a, d)) )
              [](badaccess -> MEM_BUS(m)) )
```

Evaluation

This is a simplified version of memory that does not take into account caching or any more complicated features like SPARC total store ordering. However this memory model is good enough for my purposes as all I need is to track addresses for the requirements relating to memory region access and this model allows me to do so in an intuitive way. This model captures enough detail for requirements PK-3 and PK-4 in appendix A.

4.4 Memory Protection Unit

The Memory Protection Unit, or Memory Management Unit (MMU), is a hardware element that watches the bus for reads and writes to memory. It maintains a list of blocked addresses and if it sees an access to a blocked address it will raise an exception called a Memory Fault.

4.4.1 Real System

When using SPARC, system designers are free to use the most appropriate Memory Management Unit for their application. If desired, they can use no MMU at all. However in the SPARC v8 docs they specify a reference MMU that is appropriate for the majority of applications. [28] Leon3 implements this SPARC V8 reference MMU [35]. This MMU is responsible for the following:

- Maintains Execute, Read and Write access information for each memory area so a process cannot access the address space of another process.
- Performs address translation from virtual addresses to physical addresses in memory so that a process with a large memory footprint does not need to be located in contiguous sections of memory. Pages are 4K-bytes in size. 32-bit virtual addresses are translated to 36-bit physical address, allowing for an effective 64-gigabyte physical address space. [28]
- If a process attempts to write to an area it is not permitted to, translation will fail as there will be no Page Table Entry found in the set of Page Table Descriptors and an exception is generated. (This is the memfault as described above.)
- If any other type of error occurs, the information is stored in the Fault Status Register and an exception is raised to the processor.

4.4.2 My MMU Model

Description

My Memory Management unit's main purpose is to watch the bus for addresses that violate its blocked set. This is captured by the TRANFER choice below. The MMU maintains a piece of state called 'blocked' which is a set of addresses that are considered illegal. On the event of an illegal access, the MMU will perform a raise event which passes a memory fault interrupt to the interrupt controller for processing. The CPU has supervisor level instructions it can use to change this blocked set and they interface with the MMU via the mmuBlock and mmuUnblock events.

CSPM code

```
mmu_init= {aX}
channel raise, raiseFailed : InTag  --synced with IRQ
channel mmuBlock, mmuUnBlock : Mem_Region    --synced with CPU

MMU(blocked)
=   tick -> ( TRANSFER(blocked)
           [] BLOCK_MEM_REGION(blocked)
           [] UNBLOCK_MEM_REGION(blocked)
           [] MMU(blocked) )

TRANSFER(blocked)
= bus?dir?addr -> ( if member(addr, blocked)
                    then ( badaccess -> raise!memfault -> MMU(blocked) )
                    else ( mmuOK -> MMU(blocked) ) )

BLOCK_MEM_REGION(blocked)
= mmuBlock?a -> MMU(union(blocked,{a}))

UNBLOCK_MEM_REGION(blocked)
= mmuUnBlock?a -> MMU(diff(blocked,{a}))
```

Evaluation

This model of the MMU is designed to be detailed enough to capture behaviour for requirement PK-233 which allows for memory regions to be configured. This model deliberately does not model more complicated behaviour like virtual and physical memory. The model matches the high level behaviour of the SPARC v8 reference MMU and thus I am satisfied with the model.

4.5 Interrupt Controller

The Interrupt Controller (also know as the 'IRQ') is the hardware element that is responsible for responding to exceptions raised by other elements in the system, prioritizing them, masking them if appropriate and forwarding them to the CPU to be processed.

4.5.1 Real System

In Leon3 the interrupt controller is responsible for monitoring incoming interrupts 1-15 of the interrupt bus. Interrupt 15 has the highest priority and interrupt 1 the lowest. Each interrupt can be assigned one of two levels, 0 and 1. Level 1 has a higher priorty than level 0. When requested by the CPU, the highest priority interrupt from level 1 will be forwarded. Only when there are no interrupts available on level 1 will interrupts

from level 0 will be forwarded.

Interrupts are masked using an interrupt mask register. This happens in each processor seperately if the system is multi-core.

The interrupt controller maintains a register called "pending" where pending interrupts are recorded. When the processor acknowledges the interrupt the corresponding bit in "pending" is cleared. There is also a "force" register with similar functionality for raising high priority interrupts. In the SPARC CPU, the EnableTraps bit in the Program Status Register must be set to 1 for traps to be acknowledged by the CPU.

Leon3 implements the SPARC Default Trap Model. [35] This means that all exceptions induced by an instruction occur before any program visible state has been changed.

4.6 CPU

4.6.1 Real System

The Leon3 CPU is based on the Integer Unit of the SPARC V8 specification [35] which contains the general-purpose registers and controls the overall operation of a SPARC processor. The LEON3 CPU is designed for low complexity and high performance. It has a seperate instruction cache and data caches (8 kByte and 4 kByte respectively), with an IEEE-754 floating point unit and support for 8 register windows. Instructions are processed in a 7 stage instruction pipeline:

1. FETCH - The instruction is fetched from the intruction cache.
2. DECODE - Instruction is decoded. Branch target addresses are generated.
3. REGISTER ACCESS - Operands are read from the register file.
4. EXECUTE - For reads and writes the address is generated. Shifts and logical operations are performed.
5. MEMORY - Any results from previous stage are written to the data cache. Data is alligned in the caches for cache reads.
6. EXCEPTION - Any pending Interrupts are resolved. (Because SPARC only handles one interrupt at a time, systems based on separate data and instruction caches with separate MMUs are required to present a single MMU interface to the operating system. [28] This would be relevent if I was looking at multiple cores for my thesis which I am not.)
7. WRITE - For any ALU, shift, logical or memory operations the results are written back to the register file.

4.6.2 My CPU Model

Description

The CPU is the most complicated model in my system. To capture its behavior effectively, I defined an instruction set that can be fed into the CPU by syncing on the 'fetch' event. (See appendix B for full code listing of CPU.)

```
--Instruction format is fe.me.nopInst.aDC.dDC.jDC
--where 'me' is used to identify the current code segment process. EG: 'kernel', 'partion1' etc.
--nopInst is the operation
--aDC stands for 'Address Dont Care'. Address goes here if appropriate.
--dDC stands for 'Data Dont Care'. Data goes here if appropriate.
--jDC stands for 'Jump Dont Care'. Code segment ID goes here if appropriate.

datatype Op = nopInst           --do nothing
            | writeInst         --write data to address
            | readInst          --read data from address
            | mmuBlockInst      --reprogram MMU, block address. This is a supervisor instruction.
            | mmuUnBlockInst    --reprogram MMU, unblock address. This is a supervisor instruction.
            | setSupModeInst    --set mode to supervisor. This is a supervisor instruction.
            | setUsrModeInst    --set mode to user.
            | jumpInst          --jump to specified code segment
            | returnInst        --load stored PC to return to prior code segment.
```

Every clock tick the CPU checks the Interrupt Controller for pending interrupts and if they are found execution will jump to the appropriate interrupt handler.

```
--if traps are disabled, skip this step.
CHECK_IRQ_ENABLED(currLoc, currSupBit, lastLoc, lastSupBit, TrapsEnabled)
= ( if TrapsEnabled == false
    then ( irqFwdDisabled -> FETCH(currLoc, currSupBit, lastLoc, lastSupBit, TrapsEnabled) )
    else ( CHECK_INTERRUPTS(currLoc, currSupBit, lastLoc, lastSupBit, TrapsEnabled) )
  )

CHECK_INTERRUPTS(currLoc, currSupBit, lastLoc, lastSupBit, TrapsEnabled)
= (noPendingIRQs -> FETCH(currLoc, currSupBit, lastLoc, lastSupBit, TrapsEnabled) )

  [] ( irqForward?tag ->
     ( --this block of code is equivilent to trap table (matching interrupts to handler code)
     if tag == memfault
     then ( HANDLE(currLoc, currSupBit, lastLoc, lastSupBit, TrapsEnabled, mf_handler) )
     else ( if tag == supInst
            then ( HANDLE(currLoc, currSupBit, lastLoc, lastSupBit, TrapsEnabled, supI_handler) )
            else ( if tag == schedulerInterrupt
                   then ( HANDLE(currLoc, currSupBit, lastLoc, lastSupBit, TrapsEnabled, scheduler) )
                   else( BADINTERRUPTTAG -> STOP )
                 )
          )
     )
     )

--fetch the specified code segment and disable traps.
HANDLE(currLoc, currSupBit, lastLoc, lastSupBit, TrapsEnabled,handler)
=  FETCH(handler, true, currLoc, currSupBit, false)
```

CPU state consists of the Traps Enabled bit, the Supervisor bit, the location of the Program Counter. The CPU also has the capability to record the previous Program Counter and Supervisor bit value which is used for operations like returning from an interrupt handler.

```
CPU(currLoc, currSupBit, lastLoc, lastSupBit, TrapsEnabled)--(currLoc, currSupBit, lastLoc, lastSupBit)
= tick -> SYNC_DEVICES(currLoc, currSupBit, lastLoc, lastSupBit, TrapsEnabled)
```

Before executing an instruction, the CPU will check if it is a user or supervisor instruction, and raise an exception if a supervisor instruction is about to be run in user mode. Otherwise it will perform the operation.

```
CHECK_SUP_BIT_INST(currLoc, currSupBit, lastLoc, lastSupBit, TrapsEnabled, op.jmp.addr.dat )
=  (if currSupBit == false
   then (if member(op, supervisor_instructions)
         then ( raise!supInst -> CPU(currLoc, currSupBit, lastLoc, lastSupBit, TrapsEnabled) )
         else EXEC_RETURN_INST(currLoc, currSupBit, lastLoc, lastSupBit, TrapsEnabled, op.jmp.addr.dat ) )
  else EXEC_RETURN_INST(currLoc, currSupBit, lastLoc, lastSupBit, TrapsEnabled, op.jmp.addr.dat ) )
```

Evaluation

This model was one of the most iterated on models that I developed. This model started as a very simple process that wrote to memory and progressively became more complicated. I implemented and then chose to leave out features like the 7 stage pipeline as it ended up being extremely complicated to debug or draw conclusions from as I was looking at traces for 7 instructions at once. After a number of iterations I realized the most fundamental thing to model was the order that the CPU performs operations and that having too high a granularity made the model too cumbersome to match to Leon3.

My CPU model performs all of the functions that I need to test the requirements and is accurate to Leon3 and SPARC. As such I am satisfied with the model.

4.6.3 My Interrupt Controller Model

Description

The interrupt controller has two interfaces, one for recieving interrupts with a raise event and the other for forwarding interrupts to the CPU with a forward event. My model of the interrupt controller is a process with a list of tuples as state. This list of tuples corresponds to the 'pending' bit vector as described in the Leon3 interrupt controller, where each interrupt is assigned a bit which is 1 when the interrupt has been raised and 0 when it has not. This queue is ordered according to how interrupts are prioritized by Leon3 / SPARC.

To reduce state space, interrupts from peripherals and devices are handled once every clock tick by syncing devStart, devRaise, and devEnd events with the peripherals process. This simulates N interrupts arriving randomly from multiple sources any clock tick but does not explode the state space by trying to investigate every possible combination of every raisable interrupt every single clock tick.

CSPM code

```
datatype InTag = memfault | supInst | timer1Interrupt | timer2Interrupt | schedulerInterrupt
interruptVector = < (supInst, false),
                    (memfault, false),
                    (schedulerInterrupt, false),
                    (timer1Interrupt, false),
                    (timer2Interrupt, false) > --(list in order of priority high to low)

channel irqForward, devRaise : InTag --synced with CPU
channel noPendingIRQs      --synced with CPU

channel devStart, devEnd

IRQ(vector)
=  ( RAISE(vector) )
   [] ( FORWARD_TO_CPU(vector) )
   [] ( devStart -> PROCESS_DEVICE_INTERRUPTS(vector) )

RAISE(vector) = raise?(x) -> IRQ(raise_interrupt(x, vector) )
raise_interrupt(tag, vector) = set_bit(tag, <>, vector)

PROCESS_DEVICE_INTERRUPTS(vector) --Every clock tick, process all peripheral interrupts.
=  (  devEnd -> IRQ(vector) )
   [] ( DEV_INTERRUPT(vector) )

DEV_INTERRUPT(vector)
= devRaise?x  -> ( (DEV_INTERRUPT(raise_interrupt(x, vector) ))
                   [](devEnd -> IRQ(raise_interrupt(x, vector) )) )
```

```
28
29 FORWARD_TO_CPU(vector)
30  = ( if is_irq_pending(<>,vector) == false
31      then ( noPendingIRQs -> IRQ(vector) )
32      else ( irqForward!(get_next_tag(<>,vector)) ->
33             IRQ(clear_bit(get_next_tag(<>,vector), <>, vector)) ) )
```

Evaluation

This model was designed to be used to reason about requirement PK-58. This interrupt controller model captures the main aspects of Leon3 and Sparc that I am interested in, which is the following:

- Multiple Interrupts coming in during a single clock tick.
- Multiple raises of the same interrupt will not flood the interrupt queue.
- Interrupts being prioritized as defined in the Sparc / Leon3 documentation.
- Like the real system starvation of a raised interrupt is possible

As such I am satisfied with the model.

4.6.4 Evaluation

4.7 Interrupt Handlers

4.7.1 Real System

Interrupt handlers always run in supervisor mode. If user trap handlers are required a record can be passed by the supervisor to the user trap handler containing the current Program Counter (PC) address, the next Program Counter (nPC) address, and any other state required for the user trap handler to be emulated by the supervisor. This satisfies the PK-69 requirement in appendix A.

4.7.2 My Model

Description

Handlers are code sections that are fetched from the CPU on the event of a trap being forwarded. When a processor jumps to a trap handler, the TrapsEnabled bit is set and no other traps can be forwarded. On a handlerReturnInst the TrapsEnabled bit is cleared.

CSPM code

```
1  MEMFAULT_HANDLER(me) =
2  --fe.me.setSupModeInst.jDC.a0.d0 -> --Instruction to set mode to supervisor
3  fe.me.nopInst.jDC.aDC.dDC ->
4  fe.me.handlerReturnInst.jDC.aDC.dDC -> --Instruction to enable IRQ forwarding, return to previous code section + mode by
5  MEMFAULT_HANDLER(me)
6
7  SUPINST_HANDLER(me) =
8  fe.me.nopInst.jDC.aDC.dDC ->
9  fe.me.handlerReturnInst.jDC.aDC.dDC -> --Instruction to enable IRQ forwarding, return to previous code section + mode by
10 SUPINST_HANDLER(me)
```

Evaluation

My interrupt handlers neatly capture the main features of the LEON3 interrupt handlers, for example being always in supervisor mode.

4.8 Peripherals

4.8.1 Real System

Peripherals are connected to the Leon architecture via the AMBA advanced peripherals bus. Each I/O port can drive a seperate interrupt line on the bus. Example of Peripherals are the General Purpose Timer Unit which communictaes over the General Purpose I/O Port. The General Purpose Timer Unit can be used to generate interrupts periodically.

4.8.2 My Peripherals Model

Description

To capture the behavior of multiple peripherals in my model I define a process called Peripherals which communicates with the Interrupt Controller. This process syncs on clock ticks and can be used to script devRaise events which raise an external device interrupt when synced with the Interrupt Controller model.

CSPM code

```
--Causes two timers to interrupt every third clock tick. In the event
--of multiple interrupts at once, the higher pririty is handled.
DEVICES =
DEVICES_LOOP
DEVICES_LOOP =

tick ->
devStart ->
devRaise!timer1Interrupt ->
devRaise!timer2Interrupt ->
devEnd ->

tick ->
devStart ->
devEnd ->

tick ->
devStart ->
devEnd ->
DEVICES_LOOP
```

Evaluation

This peripherals model is ideal for faking non-deterministic interrupts deterministically and in such a way that minimizes state space. It matches how the Leon3 AMBA interrupt bus performs and thus I am very satisfied with this model.

4.9 System Integrator

4.9.1 Real System

The system integrator is responsible for managing the initial set up of the partition kernel system. It carries out this role by taking configuration defined in an XML schema and using it to configure the virtual hardware of the system. The system integrator is responsible for configuring the timing requirements, memory configuration and external interface requirements of each partition based on what the programmer specifies in the XML configuration file.

4.9.2 My Model

While I do not have an explicit system integrator in my model my model emulates this by using a scenario driven approach. For defining the setup of memory regions to partitions as described in the requirements PK-3 and PK-233 I can simply use instructions to script the Memory Management Unit for each partition and encode the relevent inforation of a system integrator that way. For example:

```
KERNEL_INIT_CODE(owner) =
fe.me.mmuBlockInst.jDC.a0.dDC ->  --block a0
fe.me.mmuBlockInst.jDC.a1.dDC ->  --block a1
fe.me.mmuUnBlockInst.jDC.a2.dDC -> --unblock a2
KERNEL_INIT_CODE(owner)
```

4.10 Scheduler

4.10.1 Real System

Partitions are scheduled on a cyclic basis. The kernel maintains a major time frame for each plan which is repeated throughout the plans operation. This major time frame is split between partitions with each one executing one at a time for a limited time which can vary between partitions. This allows partitions uninterrupted access to all the hardware resources in the system and keeps the system deterministic.

4.10.2 My Model

The scheduler is in two parts. I make used of the clock to trigger a scheduler interrupt at fixed increments.

```
--see peripherals section
tick ->
devStart ->
devRaise!schedulerInterrupt ->
devEnd ->
```

When the scheduler handler is called it continues executing from where it last was because as it is a CSP process it cannot 'go back' on itself. I take advantange of this behaviour to use it to jump between partitions on alternate calls.

```
SCHEDULER_HANDLER(me) =
fe.me.mmuBlockInst.jDC.a0.dDC -> --block a0
fe.me.mmuUnBlockInst.jDC.a1.dDC -> --unblock a1
fe.me.mmuBlockInst.jDC.a2.dDC -> --block a2
fe.me.setUsrModeInst.jDC.aDC.dDC ->
fe.me.jumpInst.partition1.aDC.dDC -> --Instruction to jump to Partition1


```

```
fe.me.mmuBlockInst.jDC.a0.dDC ->  --block a0
fe.me.mmuBlockInst.jDC.a1.dDC ->  --block a1
fe.me.mmuUnBlockInst.jDC.a2.dDC -> --unblock a2
fe.me.setUsrModeInst.jDC.aDC.dDC ->
fe.me.jumpInst.partition2.aDC.dDC -> --Instruction to jump to Partition2
SCHEDULER_HANDLER(me)
```

Evaluation

The scheduler that I implemented is quite simple. A Major frame is approximated but a limitation is the timing overhead of the blocks and unblocks and jump instructions in my model. This is something I would have liked to expand on if I had more time.

4.11 Partitions

4.11.1 Real System

Partitions as described in the ESA requirements document are virtual machines controlled by the seperation kernel. [11] The seperation between the kernel and partitions is provided by the User / Supervisor mode of the processor. (See requirement PK-230 in appendix A)

Each partition has a predefined memory area allocated to it. Memory isolation is ensured by dissallowing at minimum write access to memory areas outside the control of a partition. Memory areas for each partition are defined in configuration used by the system integrator. (See requirement PK-3 in appendix A) The context of a partition is managed by the virtual CPU on which the partition is running. This context information includes the memory areas allocated to partitions and all virtual traps / interrupts etc allocated to the partition.

Partitions can be in one of 4 modes.

1. READY - The partition is ready to start execution at its next available timeslot
2. RUNNING - The partition is currently executing in its timeslot. Context such as virtual traps / interrupts is saved as partitions enter and leave this state.
3. ACTIVE - A macro state composed of READY and RUNNING. Whenever active state is started, the partition can either start *warm* or *cold* which can be used to indicate which application data should be kept or discarded.
4. STOPPED - The partition has allocated timeslots but is not executing in them. A second partition can restart this partition with either a *cold* or *warm* starting type.

4.11.2 My Model

Description

Partitions in my model are code segments that are called by the kernel and handler. They capture a core aspect of a partition, that it is an autonomous entity that interfaces with the CPU at scheduled times.

CSPM code

```
PARTITION_1(me) =
fe.me.nopInst.jDC.aDC.dDC ->
PARTITION_1_DONE(me)
PARTITION_1_DONE(me) =
fe.me.nopInst.jDC.aDC.dDC ->
PARTITION_1_DONE(me)


PARTITION_2(me) =
PARTITION_2_DONE(me)
PARTITION_2_DONE(me) =
fe.me.nopInst.jDC.aDC.dDC ->
fe.me.writeInst.(jDC)?(x).(d0) -> --Attempt to write to all possible addresses
PARTITION_2_DONE(me)
```

Evaluation

I was dissapointed I did not get a chance to implement the Partition modes in my model, however it was not a priority for my thesis and could be easily extended if required by adding state to the Partition process.

Chapter 5

Experimental Procedure

5.1 Introduction

In this section I will describe the experimental procedure I underwent to test the model described in the previous section. As this model may eventually be linked to the Xtratum source code via a formal proof, the purpose of these experiments is to demonstrate a way to run queries against the model. These tests are all built up using FDR3's deadlock checking features and as such these experiments will start simply and gradually build in complexity with an aim to convince the reader of the power of this testing system.

5.2 Experiment 1 - Illegal Write Detection

5.2.1 Motivation:

The purpose of this experiment is to demonstrate using the deadlock testing capabilities of FDR3 to design an approach to detect illegal writes to memory in our model.

5.2.2 Design:

I designed a scenario which was designed to mimic the kernel attempting to do a write to an address. My independent variable of this experiment was the address and I chose to examine three possibilities:

- Legal write - This acts as a control for the experiment. I expect this to succeed.
- Illegal write - The kernel attempts to write to a forbidden address aX. I expect to get a memory fault interrupt.
- All possible writes - I use the '?' CSPM operator to denote a choice of address. I expect this to get a memory fault interrupt when aX is chosen.

To look out for the memory fault interrupt I will be debuting a checker process. This is an idea that I will be using throughout these experiments. Because I sync all my models with the clock, the checker only needs to sync on ticks, where it will go back and behave like itself, and on the raising of memfaults, in which case it will transition into CSP's STOP process, and deliberately deadlock. As we can use FDR3 to run assertions that deadlock has not occured, we can use these assertions to reason about high level properties of the system.

```
1 --Deadlock if a memfault is detected.
2 CHECKER = ( ( tick -> CHECKER )
3         []
```

```
        ( (raise.memfault -> CHECKER_TEST_FAILED -> STOP) ) )

assert SYSTEM :[deadlock free [F]]
```

5.2.3 Assumptions and Limitations:

I designed the experiment by having a kernel try and write to an imaginary address that is illegal to every entity called aX. This is not realistic as the kernel should have access to the whole memory space. If I was doing this experiment again to make it more realistic I would have the illegal write be a Partition writing to the memory space of another Partition as may happen in the real Xtratum system.

5.2.4 Code Listing:

```
include "../system.csp"

-------------
--SCENARIO:
-------------

--define kernel and handlers
KERNEL_INIT_CODE(me) =
fe.me.nopInst.jDC.aDC.dDC ->
fe.me.mmuUnBlockInst.jDC.a0.dDC ->
fe.me.nopInst.jDC.aDC.dDC ->
KERNEL_LOOP(me)

KERNEL_LOOP(me) =
--This write instruction is our independent variable in experiment 1
fe.me.writeInst.jDC.a0.d0 ->    --do a good write
--fe.me.writeInst.jDC.aX.d0     --do a bad write
--fe.me.writeInst.(jDC)?x.(d0) -> --do all possible writes
KERNEL_LOOP(me)

MEMFAULT_HANDLER(me) =
fe.me.nopInst.jDC.aDC.dDC ->
fe.me.handlerReturnInst.jDC.aDC.dDC ->
MEMFAULT_HANDLER(me)

SUPINST_HANDLER(me) =
fe.me.nopInst.jDC.aDC.dDC ->
fe.me.handlerReturnInst.jDC.aDC.dDC ->
SUPINST_HANDLER(me)

-------------
--HARDWARE:
-------------

--Put all entities from system.csp in paralell
CODE0 = KERNEL_INIT_CODE(kernel)
CODE3 = CODE0 [| {| |} |] MEMFAULT_HANDLER(mf_handler)
CODE4 = CODE3 [| {| |} |] SUPINST_HANDLER(supI_handler )
CODE = CODE4
INTERNAL_SYSTEM0 = CPU(kernel, true, kernel, true, true)
INTERNAL_SYSTEM1 = INTERNAL_SYSTEM0 [| {| tick, read, bus, write |} |] MEM_BUS (mem_init)
INTERNAL_SYSTEM2 = INTERNAL_SYSTEM1
                 [| {| tick, bus,
                   badaccess, mmuOK,
                   mmuBlock, mmuUnBlock |} |] MMU(mmu_init)
INTERNAL_SYSTEM3 = INTERNAL_SYSTEM2
                 [| {| noPendingIRQs, raise,
                 irqForward, devStart,
                  devRaise, devEnd|} |] IRQ(interruptVector)
INTERNAL_SYSTEM5 = INTERNAL_SYSTEM3 [| {| tick, fe |} |] CHECKER
INTERNAL_SYSTEM6 = INTERNAL_SYSTEM5 [| {| fe, SCENARIO_COMPLETED |} |] CODE
SYSTEM = INTERNAL_SYSTEM6 [| {| tick |} |] CLOCK
```

5.2.5 Test Results:

- Legal Write - Deadlock assertion passed
- Illegal Write - Deadlock assertion failed
- All possible writes - Deadlock assertion failed

5.2.6 Traces:

FDR3 allows us to graphically visualize the traces of the simulation. For Experiment 1 I was able to generate traces for all scenarios.

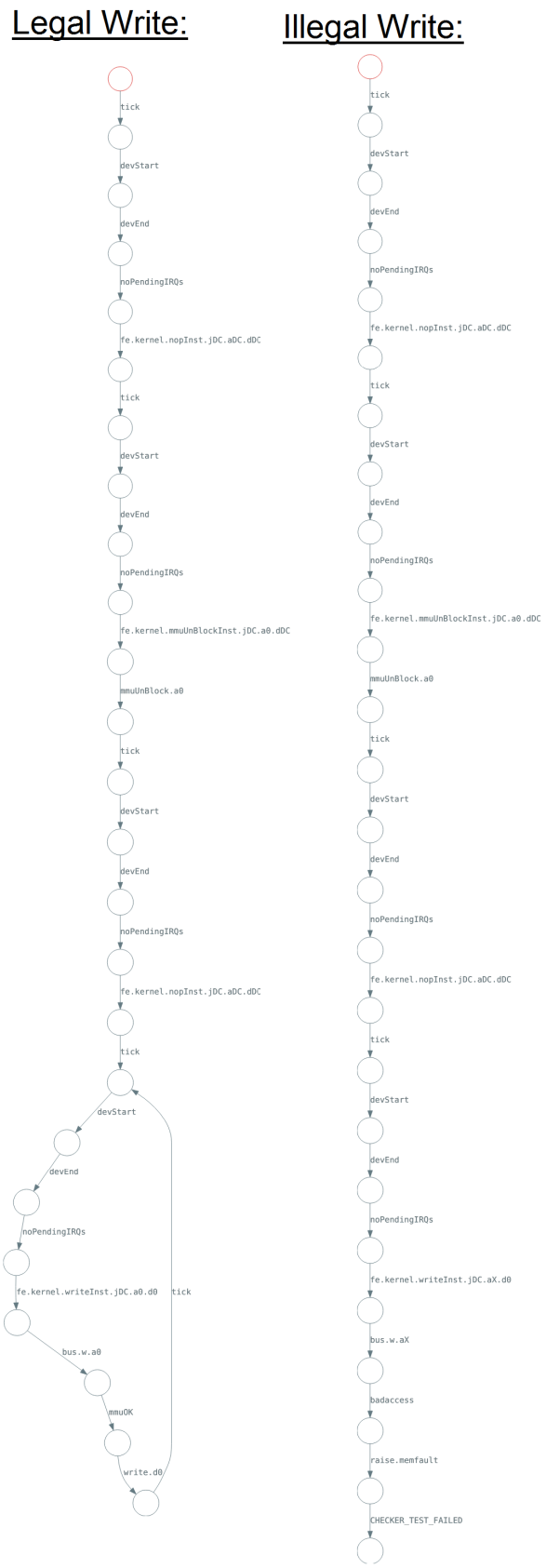
Legal Write:
tick
devStart
devEnd
noPendingIRQs
fe.kernel.nopInst.jDC.aDC.dDC
tick
devStart
devEnd
noPendingIRQs
fe.kernel.mmuUnBlockInst.jDC.a0.dDC
mmuUnBlock.a0
tick
devStart
devEnd
noPendingIRQs
fe.kernel.nopInst.jDC.aDC.dDC
tick
devStart
devEnd
noPendingIRQs
fe.kernel.writeInst.jDC.a0.d0
tick
bus.w.a0
mmuOK
write.d0
Illegal Write:
tick
devStart
devEnd
noPendingIRQs
fe.kernel.nopInst.jDC.aDC.dDC
tick
devStart
devEnd
noPendingIRQs
fe.kernel.mmuUnBlockInst.jDC.a0.dDC
mmuUnBlock.a0
tick
devStart
devEnd
noPendingIRQs
fe.kernel.nopInst.jDC.aDC.dDC
tick
devStart
devEnd
noPendingIRQs
fe.kernel.writeInst.jDC.aX.d0
bus.w.aX
badaccess
raise.memfault
CHECKER_TEST_FAILED

Figure 5.1:

All Possible Writes:

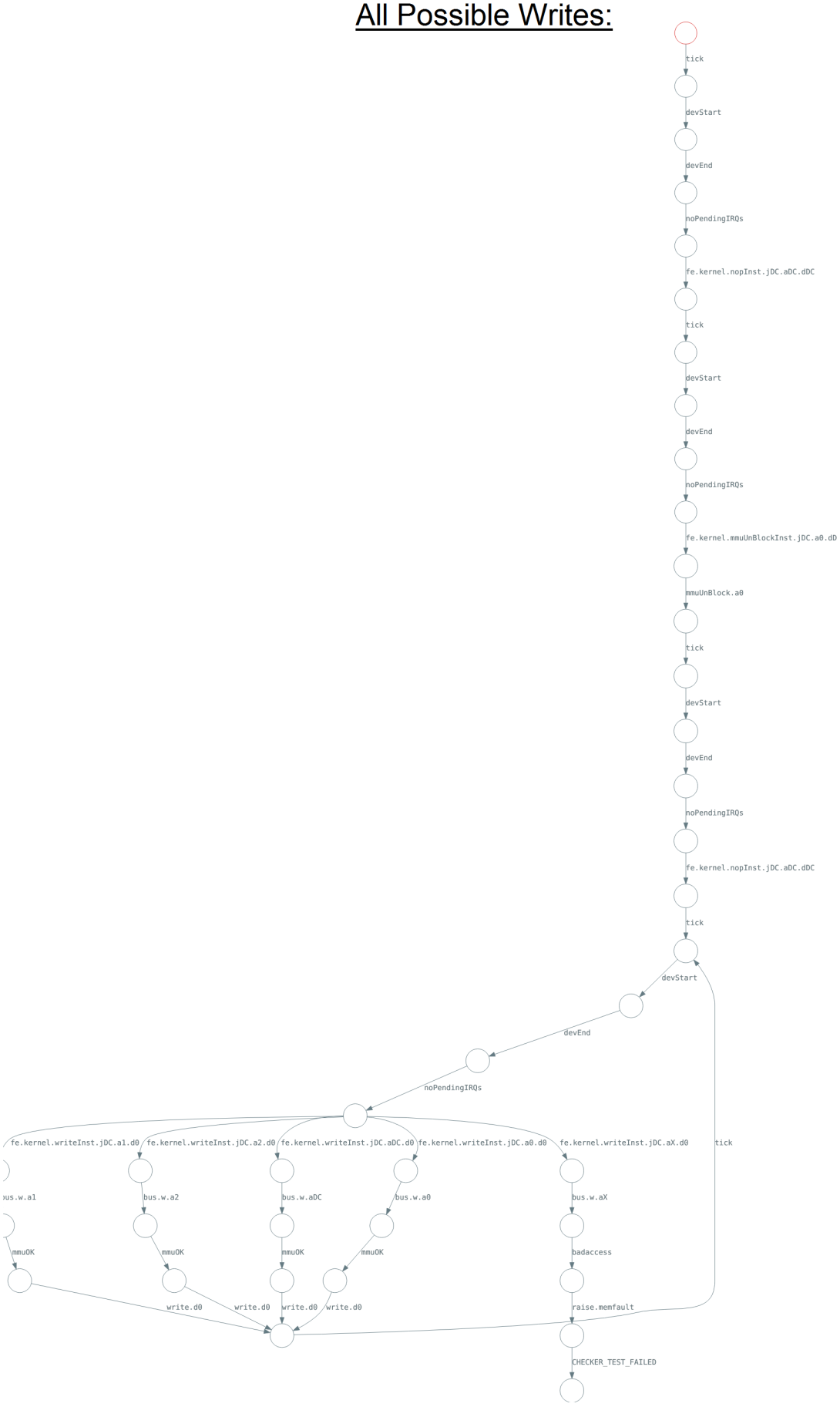

Figure 5.2:

5.2.7 Analysis

- Legal Write - The trace for the legal write has a distinctive loop shape that indicates it is a deterministic and non-deadlocking system.
- Illegal Write - The trace for the illegal write is cut off abruptly when the illegal write happens and the memfault is raised, which triggers the deadlock.
- All possible Writes - We can distictly see that writes to a0, a1, and a2 do not deadlock, but when aX is chosen no more execution paths are possible.

5.2.8 Conclusion

We have demonstrated using FDR3's deadlock test capability to build up a more complicated test using a checker process and shown that this test works for all writes.

5.3 Experiment 2 - User / Supervisor Detection

5.3.1 Motivation

The purpose of this experiment is to detect whether a supervisor mode instruction can ever be executed in user mode for high level entities in the system such as the kernel, partitions, and interrupt handlers. This should demonstrate how the requirement PK-230 ("Partitions shall be executed in user mode") might be checked if the Xtratum source code was linked to the model.

5.3.2 Design

For this experiment we will define a scenario with a kernel, an interrupt handler, and a partition. Our independent variable will be varying which of these entities attempts the supervisor instruction. We will use a checker process as in experiment 1 except this time we will look out for 'supInst' interrupts and deadlock on their detection. To manage state space, I do not think it is necessary to perform all possible instructions in this case, only the known supervisor one. My expectation of the results is as follows:

- Kernel - This is the control of the experiment. I do not expect a supervisor instruction exception as the kernel always runs in supervisor mode.
- Handler - I do not expect this to trigger a supervisor instruction exception as handlers are part of the kernel and run in supervisor mode.
- Partition - I expect this to trigger a supervisior instruction exception as partitions should always execute in user mode.

5.3.3 Assumptions and Limitations

An assumption in these experiments is that our model matches the behaviour of Xtratum running on Leon3.

I am assuming that my simulation is set up correctly apart from the three areas I wish to test: kernel, partition, handler.

5.3.4 Code Listing

```
1  include "system.csp"
2
3  KERNEL_INIT_CODE(me) =
4  fe.me.setSupModeInst.jDC.a0.d0 -> --Instruction to set mode to supervisor for part A
5  fe.me.writeInst.jDC.aX.d0 ->  -- Trigger memfault handler for part B
6  KERNEL_LOOP(me)
7  KERNEL_LOOP(me) =
8  fe.me.nopInst.jDC.aDC.dDC ->
9  KERNEL_LOOP(me)
10
11 PARTITION_1(me) =
12 fe.me.nopInst.jDC.aDC.dDC ->
13 fe.me.setSupModeInst.jDC.a0.d0 -> --Instruction to set mode to supervisor for part C
14 PARTITION_1_DONE(me)
15 PARTITION_1_DONE(me) =
16 fe.me.nopInst.jDC.aDC.dDC ->
17 PARTITION_1_DONE(me)
18
19
20 PARTITION_2(me) =
21 fe.me.nopInst.jDC.aDC.dDC ->
```

```
22 PARTITION_2_DONE(me)
23 PARTITION_2_DONE(me) =
24 fe.me.nopInst.jDC.aDC.dDC ->
25 PARTITION_2_DONE(me)
26
27 --jump between partitions each time handler is called.
28 SCHEDULER_HANDLER(me) =
29 fe.me.setUsrModeInst.jDC.aDC.dDC ->
30 fe.me.jumpInst.partition1.aDC.dDC ->
31 fe.me.setUsrModeInst.jDC.aDC.dDC ->
32 fe.me.jumpInst.partition2.aDC.dDC ->
33 SCHEDULER_HANDLER(me)
34
35 MEMFAULT_HANDLER(me) =
36 --fe.me.setSupModeInst.jDC.a0.d0 -> --Instruction to set mode to supervisor for part B
37 fe.me.nopInst.jDC.aDC.dDC ->
38 fe.me.handlerReturnInst.jDC.aDC.dDC ->
39 MEMFAULT_HANDLER(me)
40
41 SUPINST_HANDLER(me) =
42 fe.me.nopInst.jDC.aDC.dDC ->
43 fe.me.handlerReturnInst.jDC.aDC.dDC ->
44 SUPINST_HANDLER(me)
45
46
47 DEVICES = DEVICES_LOOP
48 DEVICES_LOOP =
49 --wait 10 clock ticks
50 tick ->
51 devStart ->
52 devEnd ->
53
54 --above block repeats 10 times, omitted here.
55
56 tick ->
57 devStart ->
58 devRaise!schedulerInterrupt ->
59 devEnd ->
60 DEVICES_LOOP
61
62
63 ------------
64 -CHECKER
65 ------------
66
67 CHECKER = ( ( tick -> CHECKER )
68           []
69           ( ( (raise.supInst -> CHECKER_TEST_FAILED -> STOP) ) ) )
```

5.3.5 Test Results

- Kernel - Deadlock assertion passed as expected
- Handler - Deadlock assertion passed as expected
- Partition - Deadlock assertion failed as expected

5.3.6 Analysis

- Kernel - As the Kernel starts in supervisor mode the supervisor instruction is allowed to be executed.
- Handler - When the illegal write is triggered and the memfault handler is called. The supervisor bit is set and all interrupts are disabled by setting the EnableTraps bit in the CPU. When the handler tries to execute a supervisor it is allowed as expected.

- Partition - The scheduler handler operates in supervisor mode like any other handler and is responsible for configuring the memory management unit for the partition. The last thing it does before starting a partition is set the CPU to user mode. This is what causes the supInst exception and subsequent deadlock.

5.3.7 Conclusion

I was able to demonstrate that in my simulation that a supervisor instruction cannot be performed while the CPU is in user mode for the three abstract entities kernel, partition and handler, which is strong evidence that my model satisfies requirement PK-230.

5.4 Experiment 3 - User / Supervisor Detection - Expanding the State Space

5.4.1 Motivation

The purpose of this experiment is to re-do my previous experiment, except this time instead of using one supervisor instruction I will attempt to run a choice of all possible instructions twice in a row. This will cause the number of possible choices to expand to N squared, where N is the number of instructions. The purpose of this experiment is:

- to demonstrate how in my model the programmer can easily expand the state space to more rigorously examine an area of interest.
- to gather more evidence that my model satisfies requirement PK-230.
- to uncover any possible corner cases I may have missed in my previous experiment.

5.4.2 Design

This experiment is designed extremely similarly to experiment 2. We have a kernel, a handler and a partition in the scenario and we wish to detect if there has been a supInst interrupt which is triggered by a supervisor instruction being executed in user mode. We have a checker process synced with the model that will deadlock if it detects this.

My expectations for this experiment are:

- Kernel - I expect this to pass the deadlock assertion.
- Handler - I expect this to pass the deadlock assertion.
- Partition - I expect this to fail the deadlock assertion.

5.4.3 Assumptions and Limitations

As in the other experiments I am assuming that my model is an accurate simulation of the Xtratum / Leon3 system.

5.4.4 Code Listing:

```
include "system.csp"

KERNEL_INIT_CODE(me) =
fe.me?(x).(jDC).(aDC).(dDC) -> --run all possible instructions for Part A
fe.me?(x).(jDC).(aDC).(dDC) -> --run all possible instructions for Part A
fe.me.writeInst.jDC.aX.d0 ->  -- Trigger memfault handler for part B
KERNEL_LOOP(me)
KERNEL_LOOP(me) =
fe.me.nopInst.jDC.aDC.dDC ->
KERNEL_LOOP(me)

PARTITION_1(me) =
fe.me.nopInst.jDC.aDC.dDC ->
--fe.me?(x).(jDC).(aDC).(dDC) -> --run all possible instructions for Part C
--fe.me?(x).(jDC).(aDC).(dDC) -> --run all possible instructions for Part C
PARTITION_1_DONE(me)
PARTITION_1_DONE(me) =
fe.me.nopInst.jDC.aDC.dDC ->
PARTITION_1_DONE(me)


```

```
PARTITION_2(me) =
fe.me.nopInst.jDC.aDC.dDC ->
PARTITION_2_DONE(me)
PARTITION_2_DONE(me) =
fe.me.nopInst.jDC.aDC.dDC ->
PARTITION_2_DONE(me)

--jump between partitions each time handler is called.
SCHEDULER_HANDLER(me) =
fe.me.setUsrModeInst.jDC.aDC.dDC ->
fe.me.jumpInst.partition1.aDC.dDC ->
fe.me.setUsrModeInst.jDC.aDC.dDC ->
fe.me.jumpInst.partition2.aDC.dDC ->
SCHEDULER_HANDLER(me)

MEMFAULT_HANDLER(me) =
--fe.me?(x).(jDC).(aDC).(dDC) -> --run all possible instructions for Part B
--fe.me?(x).(jDC).(aDC).(dDC) -> --run all possible instructions for Part B
fe.me.nopInst.jDC.aDC.dDC ->
fe.me.handlerReturnInst.jDC.aDC.dDC ->
MEMFAULT_HANDLER(me)

SUPINST_HANDLER(me) =
fe.me.nopInst.jDC.aDC.dDC ->
fe.me.handlerReturnInst.jDC.aDC.dDC ->
SUPINST_HANDLER(me)


DEVICES = DEVICES_LOOP
DEVICES_LOOP =
--wait 10 clock ticks
tick ->
devStart ->
devEnd ->

--above block repeats 10 times, omitted here.

tick ->
devStart ->
devRaise!schedulerInterrupt ->
devEnd ->
DEVICES_LOOP


------------
-CHECKER
------------

CHECKER = ( ( tick -> CHECKER )
          []
          ( ( (raise.supInst -> CHECKER_TEST_FAILED -> STOP) ) ) )
```

5.4.5 Test Results

- Kernel - This failed the deadlock assertion which I did not expect.
- Handler - This failed the deadlock assertion which I did not expect.
- Partition - This failed the deadlock assertion which I was expecting.

5.4.6 Analysis

The deadlock in the kernel and handler is caused by a supervisor entity with all permissions being able to change to user mode, but then being prohibited from executing a supervisor instruction to change back due to now being in user mode. The idea of an instruction to change to supervisor mode is an abstraction I used for my model and does

not have a direct mapping with the Leon specification so the documentation is little help in this case. If this behavior is an issue for future experiments it can be mitigated by either writing a smarter checker test that ignores that case or modifying the model.

5.4.7 Conclusion

We have demonstrated how the model can be used to expand the state space and uncover interesting traces about our model that might not be obvious at first glance. We used this to uncover an emergent behaviour in our model that we had not anticipated and suggested possible remedies.

5.5 Experiment 4 - Investigating Partition Memory Protection

5.5.1 Motivation

The purpose of this experiment is to investigate requirements PK-3 and PK4 ("The partitioning kernel shall detect and prevent any attempt of a partition to access a memory area that is not defined as allowed for access in its configuration. ")

In this experiment we will attempt to conceptually link our memory addresses to the partitions, for example assigning partion1 to the memory region a1, partition2 to the memory region a2, etc. This is an example of taking a 'meta' view of the model and linking together concepts such as partitions and memory which are not programmatically coupled in the model.

We will also demonstrate using a more complex checker process to test for longer traces.

5.5.2 Design

In this experiment, we will have two partitions. Partition 1 is assumed to reside in memory region a1. Partition 2 represents a malicious partition which attempts write to every possible address. We use MMU block and unblock instructions in the scheduler to configure the available memory areas. We wish to find out if this memory protection scheme can effectively mitigate this malicious Partion.

We define the following checker process which is designed to deadlock on a successful write to a1.

```
CHECKER = ( (tick -> CHECKER)
          []
          ( fe.partition2.writeInst.jDC.a1.d0 -> (( write.d0 -> CHECKER_TEST_FAILED -> STOP)
                        [](tick -> CHECKER))))
```

For our control we run the scheduler without memory protection instructions and see if the checker process deadlocks. I expect the following to happen:

- Protected system - Deadlock assertion passes.
- Unprotected system - Deadlock assertion fails.

5.5.3 Assumptions and Limitations

As in the other experiments we assue our Xtratum / Leon simulation to match the real system.

5.5.4 Code Listing:

```
include "system.csp"

KERNEL_INIT_CODE(me) =
KERNEL_LOOP(me)
KERNEL_LOOP(me) =
fe.me.nopInst.jDC.aDC.dDC ->
KERNEL_LOOP(me)
```

```
8
9  PARTITION_1(me) =
10 fe.me.nopInst.jDC.aDC.dDC ->
11 PARTITION_1_DONE(me)
12 PARTITION_1_DONE(me) =
13 fe.me.nopInst.jDC.aDC.dDC ->
14 PARTITION_1_DONE(me)
15
16
17 PARTITION_2(me) =
18 fe.me.nopInst.jDC.aDC.dDC ->
19 PARTITION_2_DONE(me)
20 PARTITION_2_DONE(me) =
21 fe.me.nopInst.jDC.aDC.dDC ->
22 fe.me.writeInst.(jDC)?(x).(d0) -> --Attempt to write to all possible addresses
23 PARTITION_2_DONE(me)
24
25 --jump between partitions each time handler is called.
26 SCHEDULER_HANDLER(me) =
27 fe.me.mmuBlockInst.jDC.a0.dDC -> --block a0 - independent variable
28 fe.me.mmuUnBlockInst.jDC.a1.dDC -> --unblock a1 - independent variable
29 fe.me.mmuBlockInst.jDC.a2.dDC -> --block a2 - independent variable
30 fe.me.setUsrModeInst.jDC.aDC.dDC ->
31
32 fe.me.jumpInst.partition1.aDC.dDC ->
33 fe.me.mmuBlockInst.jDC.a0.dDC ->  --block a0  - independent variable
34 fe.me.mmuBlockInst.jDC.a1.dDC ->  --block a1  - independent variable
35 fe.me.mmuUnBlockInst.jDC.a2.dDC -> --unblock a2 - independent variable
36 fe.me.setUsrModeInst.jDC.aDC.dDC ->
37 fe.me.jumpInst.partition2.aDC.dDC ->
38 SCHEDULER_HANDLER(me)
39
40 MEMFAULT_HANDLER(me) =
41 fe.me.nopInst.jDC.aDC.dDC ->
42 fe.me.handlerReturnInst.jDC.aDC.dDC ->
43 MEMFAULT_HANDLER(me)
44
45 SUPINST_HANDLER(me) =
46 fe.me.nopInst.jDC.aDC.dDC ->
47 fe.me.handlerReturnInst.jDC.aDC.dDC ->
48 SUPINST_HANDLER(me)
49
50
51 DEVICES = DEVICES_LOOP
52 DEVICES_LOOP =
53 --wait 10 clock ticks
54 tick ->
55 devStart ->
56 devEnd ->
57
58 --above block repeats 10 times, omitted here.
59
60 tick ->
61 devStart ->
62 devRaise!schedulerInterrupt ->
63 devEnd ->
64 DEVICES_LOOP
```

5.5.5 Results

Protected memory system passed deadlock assertion as expected.

Unprotected memory system failed deadlock assertion as expected.

5.5.6 Analysis

When an illegal write was attempted on Partition 1, it was blocked by the Memory Management Unit, therby avoiding causing the checker to deadlock. As mentioned in

the System Integrator section in the previous chapter, there is no explicit configuration entity in my model for configuring partition access areas like the TSP System Integreator.However while we are in supervisor mode such as in the scheduler handler we can make use of the MMU block and unblock instructions to manage the inialization and management of memory areas without needing an explicit configuration or System Integrator entity.

5.5.7 Conclusion

We have shown a demonstration of using the model to reason about aspects of the system that are conceptually but not programmatically linked which proves that the system can be used to host 'meta-models' of higher abstract behaviour based on simpler rules. We have also demonstrated using a checker process to look for arbitrarily extendable complex traces. We have also gathered evidence that the model implements the two memory safety requirements PK-2 and PK-3, which could eventually be linked to a formal proof, however more testing would be desirable if I had more time.

Chapter 6

Conclusion

As stated in the introduction of my thesis, the goal of my thesis was to investigate the use of an intermediate formal model to capture the behavior of Xtratum running on Leon3 which could be used to capture the difficult verification task of tracking interrupts.

Using Communicating Sequential Processes and the FDR3 model checker, I created a simulation tool for Xtratum / Leon3 which captured the memory and time impact of interrupts on the hardware system. I then demonstrated this simulation tool in action by running several proof of concept experiments.

- Experiment 1 was designed to demonstrate the use of a checker process that can be used to test for the presense of traces. This involved tracking a memfault interrupt and deadlocking on its occurance.

- Experiment 2 was designed to demonstrate how the ESA requirement PK-230 that all partitions must always be executing in user mode might be checked. This was done by tracking any interrupts that occur when a supervisor instruction is executed in user mode. CSP excels as a verification tool here, allowing us to quickly test a problem that would be prohibitively comples in the real system.

- Experiment 3 was designed to further investigate the PK-230 requirement and show how CSP can be used to exhaustively check an aspect of a model without explosively growing the state space by clever use of the CSP choice operators. This experiment showed the advantages of using CSP for a project like this.

- Experiment 4 was designed to demonstrate how high level conceptual models can be projected onto my simulator. I performed an experiment which linked partitions driven by an interrupting scheduler to a memory model via an intermediate link, the configurable Memory Management Unit. I used this to show how requirements sucj as PK-3 and PK-4 might be verified in the future using my simulation tool.

Overall I hope that by documenting my experience modelling a real processor in CSP that this approach may be refined and potentially used by other researcher to link the ESA requirements to the Xtratum source code, and create a verified, open source TSP hypervisor to be used in the next generation of European space missions.

Chapter 7

Further Work

Unfortunately I ran out of time to implement all of the functionality that I wanted in the simulator. I would have liked to have done the following:

- Extend Time Aspect of Simulation - My simulation focused very heavily on memory. I would have liked to have taken a deeper look at the TSP timing constraints to allow for the high level timing features that TSP prescribes. All hardware elements in my simulation are tied to a clock, and this is something that future programmers can take advantage of.
- Health Monitoring - I was dissapointed I did not get time to implement a health monitoring unit as prescribed in the TSP specification. This entity is designed to monitor partitions for unexpected events, and thus it would extremely well suited to being modelled in CSP as it could sync on only intended events and deadlock in all others. This is made all the more tempting by the fact that something like the health monitor is so difficult to verify in other languages.
- Xtratum Source Code Linking - This is likely the next step for this verification work, it would be interesting to attempt to link the Xtratum C code to my formal model using tools like Frama-C to add pre and post-conditions to the C functions of Xtratum. By combining this with a tool I have shown I can reason about, my CSP simulation, formal models can be created and tested which capture the desired behavior of the Xtratum source code.

Appendix A

Appendix A - ESA Interrupt Requirements

Below are the requirements that are relevent to interrupts from the "Partitioning Kernel Requirements Baseline v1.2.0" document from ESA. [11]

- PK-230 / [CORE] / [SINGLE MULTI]
  - Description - Partitions shall be executed in user mode.
  - Rationale - The separation between the partitioning kernel and the application programs is provided by the Supervisor / User mode of the computer.
  - Note - none.
- PK-3 / [CORE] / [SINGLE MULTI]
  - Description - The Partitioning Kernel shall allow each partition to access its memory areas such as described in the configuration.
  - Rationale - See PK-233.
  - Note - None
- PK-4 / [CORE] / [SINGLE MULTI]
  - Description - The partitioning kernel shall detect and prevent any attempt of a partition to access a memory area that is not defined as allowed for access in its configuration.
  - Rationale - See PK-233.
  - Note - None.
- PK-233 / [CORE] / [SINGLE MULTI]
  - Description - It shall be possible to configure the access rights for each memory area accessible to a partition.

- Rationale - For Leon3/2 MMU access is configured and controlled for reading, writing and executing. For LEON2 without MMU, access is configured and controlled for writing and it is accepted that reading is allowed for all memory mapped and executing is protected by writing access configuration Note: The allowed memory areas can be volatile (RAM) or non-volatile memory (NVR) as long as their access are done without protocol (ex: non volatile Flash memory). In this case there is no difference from the kernel point of view. We assume here that a memory access "without protocol" is one that simply follows the normal RAM access process of providing address and transfer direction information, and then moving data to/from the addressed location over the standard CPU-Memory bus.
- Note - None.

- PK-9 / [CORE] / [SINGLE MULTI]
  - Description - When a partition is resumed by the partitioning kernel at the beginning of its timeslots, the said partition shall restart in the same memory context (memory allocated to the partition), CPU core registers context and FPU context (if FPU is used by the considered partition).
  - Rationale - None.
  - Note - A partition is resumed when switching from READY to RUNNING mode.

- PK-58 / [CORE] / [SINGLE MULTI]
  - Description - The Partitioning Kernel shall detect and handle all traps and interrupts.
  - Rationale - None.
  - Note - None.

- PK-557 / [CORE] / [SINGLE MULTI]
  - Description - The health monitoring shall be able to detect and handle the following health monitoring events: software traps, exception traps and synchronization error event.
  - Rationale - None.
  - Note - None.

- PK-69 / [CORE] / [SINGLE MULTI]
  - Description - he Partitioning Kernel shall allow a partition to attach a partition handler to each of the virtual traps and interrupts allocated to the said partition.
  - Rationale -

- Note - An application handler is a method that contains the code that gets executed in response to a specific event that occurs in an application (here a virtual interrupt or virtual trap). The configuration specifies for each partition which interrupts have to be routed to it. Partitioning Kernel transfers the interrupt to the partition as a virtual interrupt (software event related to the initial interrupt) if the partition subscribed to this virtual interrupt. Handlers are associated with the subscribed virtual interrupts. These handlers are executed when the partition is running. The interrupt is masked until the partition clears the virtual interrupt. Example: A partition can attach an handler to a virtual trap or interrupt by using the Virtual Trap Table of the virtual CPU.

- PK-186 / [CORE] / [SINGLE MULTI]

  - Description - The partitioning kernel shall execute the handlers of the virtual interrupts allocated to a partition according to a policy order based on the priority of the corresponding interrupts

  - Rationale - Virtual interrupts includes here the extended interrupts.

  - Note - As virtual interrupts dedicated to a partition are masked when the partition is not running, solely the virtual interrupts attached to the partition can be raised and handled in a synchronous manner. All the virtual interrupts dedicated to a partition that are triggered when the said partition is not running are not handled. The status (masked/unmasked) of the virtual interrupts is restored when the partition enters in its timeslot. The pending virtual interrupts are then handled in a specific order according to a policy based on the priority of the corresponding interrupts.

Appendix B

Appendix B - Simulator Code Listing

B.1 Simulator Code

```
1
2
3   channel tick --used for syncing each hardware component
4   CLOCK = tick -> CLOCK
5
6   ------------------------------------------------------------------------------------------------
7
8
9   channel CHECKER_TEST_FAILED
10
11
12
13  channel STACKFULL, BADINTERRUPTTAG, SCENARIO_COMPLETED
14
15
16
17  {- Memory and Bus -}
18
19  --Memory is represented by a map where Mem_Region is a key for Data
20
21  datatype CodeOwner = jDC | kernel | kernel_error | partition1 | partition2 | mf_handler
22   | supI_handler | timer1_handler | timer2_handler | scheduler | scenario_completed
23  --(DC stands for DontCare)
24
25
26  --convention - aX is always forbidden to everyone, even the kernel (for debugging)
27  datatype Mem_Region = a0 | a1 | a2 | aDC | aX
28
29
30  datatype Data = d0 | d1 | dDC
31
32  datatype Op =  nopInst | writeInst | readInst | mmuBlockInst | mmuUnBlockInst | setSupModeInst | setUsrModeInst | jumpIn
33
34
35
36  --memory is made up of 'memory regions'
37
38
39  mem_init = (| a0 => d0, a1 => d0, a2 => d0, aDC => d0, aX => d0 |)
40
41  --supervisor mode only instructions and memory regions
42  supervisor_instructions = {mmuBlockInst,mmuUnBlockInst, setSupModeInst}
43  --supervisor_mem_regions = {krnl_init,handler0}
44
45
46
```

```
47
48
49
50  datatype Bus_Dir = r | w
51  channel read, write : Data   --synced with CPU
52  channel bus : Bus_Dir.Mem_Region --synced with MMU
53  channel badaccess , mmuOK          --synced with MMU
54
55  --m is the memory map
56
57  MEM_BUS(m)
58  =   tick -> ( MEM_BUS_READ(m)
59              [] MEM_BUS_WRITE(m)
60              [] MEM_BUS(m) )
61
62  MEM_BUS_READ(m)
63  = bus.r?a -> ( ( mmuOK -> read!(mapLookup(m,a)) -> MEM_BUS(m) )
64               [] (badaccess -> MEM_BUS(m) ) )
65
66  MEM_BUS_WRITE(m)
67  =  bus.w?a -> ( ( mmuOK -> write?d -> MEM_BUS(mapUpdate(m, a, d)) )
68               [](badaccess -> MEM_BUS(m)) )
69
70
71
72
73  mmu_init= {aX}
74  channel raise, raiseFailed : InTag  --synced with IRQ
75  channel mmuBlock, mmuUnBlock : Mem_Region    --synced with CPU
76
77  MMU(blocked)
78  =   tick -> ( TRANSFER(blocked)
79            [] BLOCK_MEM_REGION(blocked)
80            [] UNBLOCK_MEM_REGION(blocked)
81            [] MMU(blocked) )
82
83  TRANSFER(blocked)
84  = bus?dir?addr -> ( if member(addr, blocked)
85                    then ( badaccess -> raise!memfault -> MMU(blocked) )
86                    else ( mmuOK -> MMU(blocked) ) )
87
88  BLOCK_MEM_REGION(blocked)
89  = mmuBlock?a -> MMU(union(blocked,{a}))
90
91  UNBLOCK_MEM_REGION(blocked)
92  = mmuUnBlock?a -> MMU(diff(blocked,{a}))
93
94
95
96
97
98
99
100
101 {- Interrupts and IRQ -}
102
103 datatype InTag = memfault | supInst | timer1Interrupt | timer2Interrupt |
104 schedulerInterrupt | scenarioDone
105 interruptVector = < (scenarioDone, false),
106                   (supInst, false),
107                   (memfault, false),
108                   (schedulerInterrupt, false),
109                   (timer1Interrupt, false),
110                   (timer2Interrupt, false) > --(list in order of priority high to low)
111
112 channel irqForward, devRaise : InTag --synced with CPU
113 channel noPendingIRQs      --synced with CPU
114
115 channel devStart, devEnd
116
117 IRQ(vector)
```

```
=  ( RAISE(vector) )
   [] ( FORWARD_TO_CPU(vector) )
   [] ( devStart -> PROCESS_DEVICE_INTERRUPTS(vector) )

RAISE(vector) = raise?(x) -> IRQ(raise_interrupt(x, vector) )
raise_interrupt(tag, vector) = set_bit(tag, <>, vector)

PROCESS_DEVICE_INTERRUPTS(vector)
=  (  devEnd -> IRQ(vector) )
   [] ( DEV_INTERRUPT(vector) )

DEV_INTERRUPT(vector)
= devRaise?x  -> ( (DEV_INTERRUPT(raise_interrupt(x, vector) ))
                  [](devEnd -> IRQ(raise_interrupt(x, vector) )) )

FORWARD_TO_CPU(vector)
 = ( if is_irq_pending(<>,vector) == false
     then ( noPendingIRQs -> IRQ(vector) )
     else ( irqForward!(get_next_tag(<>,vector)) ->
            IRQ(clear_bit(get_next_tag(<>,vector), <>, vector)) ) )









--find a tuple and set bool true while maintaining list order

set_bit(tag, sublist1, sublist2)
= if sublist2 == <>
  then sublist1
  else if head(sublist2) == (tag, false)
       then ( sublist1 ^ <(tag, true)> ^ tail(sublist2) )
       else set_bit(tag, sublist1 ^ <head(sublist2)>, tail(sublist2))


--find a tuple and set bool false while maintaining list order

clear_bit(tag, sublist1, sublist2)
= if sublist2 == <>
  then sublist1
  else if head(sublist2) == (tag, true)
       then ( sublist1 ^ <(tag, false)> ^ tail(sublist2) )
       else clear_bit(tag, sublist1 ^ <head(sublist2)>, tail(sublist2))









is_irq_pending(sublist1, sublist2)
= if sublist2 == <>
  then false
  else if is_irq_activated(head(sublist2)) == true
       then true
       else is_irq_pending(<head(sublist1)>, tail(sublist2) )

is_irq_activated( (tag,status) )
= if status == true
  then true
  else false


get_next_tag(sublist1, sublist2)
= if sublist2 == <>
  then head(sublist1) ---throws an emptylist exception.
                      --when you call this function you assume there is an IRQ present.
  else if is_irq_activated(head(sublist2)) == true
       then get_irq_tag(head(sublist2))
       else get_next_tag(<head(sublist1)>, tail(sublist2) )
```

```
189
190 get_irq_tag( (tag, status) )
191 = tag
192
193
194
195
196 {- CPU -}
197 channel irqFwdDisabled --not synced with anything, just for info
198
199 --Return popped stack. Keep the bottom element (kernel) on the bottom of the stack always
200 popped_stack(stack) = if length(stack) ==1
201                       then ( stack )
202                       else ( tail(stack) )
203
204
205
206
207
208 channel fe : CodeOwner.Op.CodeOwner.Mem_Region.Data --synced with CODE.
209
210
211
212
213
214
215
216 CPU(currLoc, currSupBit, lastLoc, lastSupBit, TrapsEnabled)--(currLoc, currSupBit, lastLoc, lastSupBit)
217 = tick -> SYNC_DEVICES(currLoc, currSupBit, lastLoc, lastSupBit, TrapsEnabled)
218
219 --used to make sending interrupts from DEVICES to IRQ more deterministic
220 SYNC_DEVICES(currLoc, currSupBit, lastLoc, lastSupBit, TrapsEnabled)
221 = devStart -> devEnd -> CHECK_IRQ_ENABLED(currLoc, currSupBit, lastLoc, lastSupBit, TrapsEnabled)
222
223 CHECK_IRQ_ENABLED(currLoc, currSupBit, lastLoc, lastSupBit, TrapsEnabled)
224 = ( if TrapsEnabled == false
225    then  ( irqFwdDisabled -> FETCH(currLoc, currSupBit, lastLoc, lastSupBit, TrapsEnabled) )
226    else ( CHECK_INTERRUPTS(currLoc, currSupBit, lastLoc, lastSupBit, TrapsEnabled) )
227  )
228
229 --channel test
230 CHECK_INTERRUPTS(currLoc, currSupBit, lastLoc, lastSupBit, TrapsEnabled)
231 = (noPendingIRQs -> FETCH(currLoc, currSupBit, lastLoc, lastSupBit, TrapsEnabled) )
232
233   [] ( irqForward?tag ->
234      ( --this block of code is equivilent to trap table (matching interrupts to handler code)
235     if tag == memfault
236     then ( HANDLE(currLoc, currSupBit, lastLoc, lastSupBit, TrapsEnabled, mf_handler) )
237     else ( if tag == supInst
238            then ( HANDLE(currLoc, currSupBit, lastLoc, lastSupBit, TrapsEnabled, supI_handler) )
239            else ( if tag == timer1Interrupt
240                   then ( HANDLE(currLoc, currSupBit, lastLoc, lastSupBit, TrapsEnabled, timer1_handler) )
241                   else ( if tag == timer2Interrupt
242                          then ( HANDLE(currLoc, currSupBit, lastLoc, lastSupBit, TrapsEnabled, timer2_handler) )
243                          else ( if tag == schedulerInterrupt
244                                 then ( HANDLE(currLoc, currSupBit, lastLoc, lastSupBit, TrapsEnabled,
scheduler) )
245                                 else( if tag == scenarioDone
246                                       then ( HANDLE(currLoc, currSupBit, lastLoc, lastSupBit, TrapsEnabled, scenario_comple
247                                       else( BADINTERRUPTTAG -> STOP )
248                                     )
249                               )
250                     )
251                )
252          )
253     )
254     )
255
256 HANDLE(currLoc, currSupBit, lastLoc, lastSupBit, TrapsEnabled,handler)
257 =  FETCH(handler, true, currLoc, currSupBit, false)
258
```

```
259
260
261
262
263
264
265
266 FETCH(currLoc, currSupBit, lastLoc, lastSupBit, TrapsEnabled)
267 = (fe.(currLoc)?inst -> CHECK_SUP_BIT_INST(currLoc, currSupBit, lastLoc, lastSupBit, TrapsEnabled, inst ) )
268
269
270 get_owner((owner, mode)) = owner
271
272 get_mode((owner, mode)) = mode
273
274
275 CHECK_SUP_BIT_INST(currLoc, currSupBit, lastLoc, lastSupBit, TrapsEnabled, op.jmp.addr.dat )
276 =  (if currSupBit == false
277    then (if member(op, supervisor_instructions)
278          then ( raise!supInst -> CPU(currLoc, currSupBit, lastLoc, lastSupBit, TrapsEnabled) )
279          else EXEC_RETURN_INST(currLoc, currSupBit, lastLoc, lastSupBit, TrapsEnabled, op.jmp.addr.dat ) )
280   else EXEC_RETURN_INST(currLoc, currSupBit, lastLoc, lastSupBit, TrapsEnabled, op.jmp.addr.dat ) )
281
282
283
284 EXEC_RETURN_INST(currLoc, currSupBit, lastLoc, lastSupBit, TrapsEnabled, op.jmp.addr.dat )
285 = if op == handlerReturnInst
286   then ( --enable IRQs and pop stack
287          CPU(lastLoc, lastSupBit, currLoc, currSupBit, true)
288        )
289   else ( EXEC_JUMP_INST(currLoc, currSupBit, lastLoc, lastSupBit, TrapsEnabled, op.jmp.addr.dat ) )
290
291
292
293
294 EXEC_JUMP_INST(currLoc, currSupBit, lastLoc, lastSupBit, TrapsEnabled, op.jmp.addr.dat )
295 = if op == jumpInst
296   then ( CPU(jmp, currSupBit, currLoc, currSupBit, true)
297        --if jmp == jDC
298          --then ( CPU(currLoc, currSupBit, lastLoc, lastSupBit, TrapsEnabled) ) --jump to DONTCARE is a nop
299          --else ( CPU(jmp, currSupBit, currLoc, currSupBit, TrapsEnabled) )
300        )
301
302   else ( EXEC_READINST(currLoc, currSupBit, lastLoc, lastSupBit, TrapsEnabled, op.jmp.addr.dat ) )
303
304
305
306
307
308
309 --replace_head(stack, jmp, trapBit ) = <(jmp, trapBit)> ^ popped_stack(stack)
310
311 --channel execute : Op.Mem_Region.Data
312
313 EXEC_READINST(currLoc, currSupBit, lastLoc, lastSupBit, TrapsEnabled, op.jmp.addr.dat )
314 = (if op == readInst
315    then (bus.r.addr -> ( (read?d -> CPU(currLoc, currSupBit, lastLoc, lastSupBit, TrapsEnabled))
316                        [] ( CPU(currLoc, currSupBit, lastLoc, lastSupBit, TrapsEnabled)) ) )
317    else (EXEC_SUPMODE_INST(currLoc, currSupBit, lastLoc, lastSupBit, TrapsEnabled, op.jmp.addr.dat )
)
318    )
319
320
321  EXEC_SUPMODE_INST(currLoc, currSupBit, lastLoc, lastSupBit, TrapsEnabled, op.jmp.addr.dat )
322  = (if op == setSupModeInst
323     then CPU(currLoc, true, lastLoc, lastSupBit, TrapsEnabled)
324     else EXEC_USRMODE_INST(currLoc, currSupBit, lastLoc, lastSupBit, TrapsEnabled, op.jmp.addr.dat ) )
325
326
327 EXEC_USRMODE_INST(currLoc, currSupBit, lastLoc, lastSupBit, TrapsEnabled, op.jmp.addr.dat )
328  = (if op == setUsrModeInst
```

```
329     then CPU(currLoc, false, lastLoc, lastSupBit, TrapsEnabled)
330     else EXEC_MMU_BLOCK_INST(currLoc, currSupBit, lastLoc, lastSupBit, TrapsEnabled, op.jmp.addr.dat ) )
331
332
333 EXEC_MMU_BLOCK_INST(currLoc, currSupBit, lastLoc, lastSupBit, TrapsEnabled, op.jmp.addr.dat )
334 = (if op == mmuBlockInst
335    then mmuBlock!addr -> CPU(currLoc, currSupBit, lastLoc, lastSupBit, TrapsEnabled)
336    else EXEC_MMU_UNBLOCK_INST(currLoc, currSupBit, lastLoc, lastSupBit, TrapsEnabled, op.jmp.addr.dat )
    )
337
338 EXEC_MMU_UNBLOCK_INST(currLoc, currSupBit, lastLoc, lastSupBit, TrapsEnabled, op.jmp.addr.dat )
339 = (if op == mmuUnBlockInst
340    then mmuUnBlock!addr -> CPU(currLoc, currSupBit, lastLoc, lastSupBit, TrapsEnabled)
341    else EXEC_WRITEINST(currLoc, currSupBit, lastLoc, lastSupBit, TrapsEnabled, op.jmp.addr.dat ) )
342
343
344 EXEC_WRITEINST(currLoc, currSupBit, lastLoc, lastSupBit, TrapsEnabled, op.jmp.addr.dat )
345 = (if op == writeInst
346    then bus.w.addr ->((write!dat -> CPU(currLoc, currSupBit, lastLoc, lastSupBit, TrapsEnabled))
347                       [](CPU(currLoc, currSupBit, lastLoc, lastSupBit, TrapsEnabled)))
348    else CPU(currLoc, currSupBit, lastLoc, lastSupBit, TrapsEnabled) )
```

Bibliography

[1] 2016. URL: http://www.gaisler.com/doc/leon3_product_sheet.pdf (visited on 05/18/2016).

[2] 2016. URL: https://docs.oracle.com/cd/E19683-01/806-5222/hwovr-17/index.html (visited on 05/17/2016).

[3] *Ada Resource Association*. URL: http://archive.adaic.com/projects/atwork/boeing.html.

[4] G. Barrett. "Model checking in practice: the T9000 virtual channel processor". In: *IEEE Transactions on Software Engineering* 21.2 (1995), pp. 69–78. ISSN: 0098-5589. DOI: 10.1109/32.345823.

[5] Christoph Baumann et al. "Computer Safety, Reliability, and Security: 28th International Conference, SAFECOMP 2009, Hamburg, Germany, September 15-18, 2009. Proceedings". In: ed. by Bettina Buth, Gerd Rabe, and Till Seyfarth. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009. Chap. Formal Verification of a Microkernel Used in Dependable Software Systems, pp. 187–200. ISBN: 978-3-642-04468-7. DOI: 10.1007/978-3-642-04468-7_16. URL: http://dx.doi.org/10.1007/978-3-642-04468-7_16.

[6] Sven Beyer et al. "Instantiating uninterpreted functional units and memory system: Functional verification of the VAMP". In: *Correct Hardware Design and Verification Methods*. Springer, 2003, pp. 51–65.

[7] E. Boiten, J. Derrick, and G. Smith. *Integrated Formal Methods: 4th International Conference, IFM 2004, Canterbury, UK, April 4-7, 2004, Proceedings*. Lecture Notes in Computer Science v. 4. Springer, 2004. ISBN: 9783540213772. URL: https://books.google.ie/books?id=hPklRYWTrXoC.

[8] Neil Brown. *An Introduction to Communicating Sequential Processes*. 2009. URL: https://chplib.wordpress.com/2009/11/16/an-introduction-to-communicating-sequential-processes/ (visited on 05/16/2016).

[9] Bettina Buth et al. "Deadlock analysis for a fault-tolerant system". In: *Algebraic Methodology and Software Technology*. Springer, 1997, pp. 60–74.

[10] Ernie Cohen et al. *Local Verification of Global Invariants in Concurrent Programs*. Tech. rep. MSR-TR-2010-9. 2010. URL: http://research.microsoft.com/apps/pubs/default.aspx?id=118664.

[11] Alexandre Cortier. *D02: Partitioning Kernel Requirements Baseline*. Tech. rep. Version 1.2.0. ESTEC Contract No. 4000111495/14/NL/GLC/al. AIRBUS Defence and Space SAS, 2015.

[12] Mads Dam et al. "Formal Verification of Information Flow Security for a Simple Arm-based Separation Kernel". In: *Proceedings of the 2013 ACM SIGSAC Conference on Computer & Communications Security*. CCS '13. Berlin, Germany: ACM, 2013, pp. 223–234. ISBN: 978-1-4503-2477-9. DOI: 10.1145/2508859.2516702. URL: http://doi.acm.org/10.1145/2508859.2516702.

[13] M. Danek et al. “Instruction set extensions for multi-threading in LEON3”. In: *Design and Diagnostics of Electronic Circuits and Systems (DDECS), 2010 IEEE 13th International Symposium on*. 2010, pp. 237–242. DOI: 10.1109/DDECS.2010.5491777.

[14] Thomas Gaska, Chris Watkin, and Yu Chen. “Integrated Modular Avionics ? Past, present, and future.” In: *IEEE Aerospace Electronic Systems* 30.9 (2015), p. 12. ISSN: 08858985. URL: http://elib.tcd.ie/login?url=http://search.ebscohost.com/login.aspx?direct=true&db=edb&AN=110859226&site=eds-live.

[15] Thomas Gibson-Robinson et al. “FDR3—A modern refinement checker for CSP”. In: *Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 2014, pp. 187–201.

[16] C. A. R. Hoare. “Communicating Sequential Processes”. In: *Commun. ACM* 21.8 (Aug. 1978), pp. 666–677. ISSN: 0001-0782. DOI: 10.1145/359576.359585. URL: http://doi.acm.org/10.1145/359576.359585.

[17] Gerwin Klein et al. “seL4: Formal Verification of an OS Kernel”. In: *Proceedings of the ACM SIGOPS 22Nd Symposium on Operating Systems Principles*. SOSP ’09. Big Sky, Montana, USA: ACM, 2009, pp. 207–220. ISBN: 978-1-60558-752-3. DOI: 10.1145/1629575.1629596. URL: http://doi.acm.org/10.1145/1629575.1629596.

[18] D. Leinenbach, W. Paul, and E. Petrova. “Towards the formal verification of a C0 compiler: code generation and implementation correctness”. In: *Software Engineering and Formal Methods, 2005. SEFM 2005. Third IEEE International Conference on*. 2005, pp. 2–11. DOI: 10.1109/SEFM.2005.51.

[19] Dirk Leinenbach and Elena Petrova. “Pervasive Compiler Verification – From Verified Programs to Verified Systems”. In: *Electronic Notes in Theoretical Computer Science* 217 (2008). Proceedings of the 3rd International Workshop on Systems Software Verication (SSV 2008), pp. 23 –40. ISSN: 1571-0661. DOI: http://dx.doi.org/10.1016/j.entcs.2008.06.040. URL: http://www.sciencedirect.com/science/article/pii/S1571066108003836.

[20] Dirk Leinenbach and Thomas Santen. “FM 2009: Formal Methods: Second World Congress, Eindhoven, The Netherlands, November 2-6, 2009. Proceedings”. In: ed. by Ana Cavalcanti and Dennis R. Dams. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009. Chap. Verifying the Microsoft Hyper-V Hypervisor with VCC, pp. 806–809. ISBN: 978-3-642-05089-3. DOI: 10.1007/978-3-642-05089-3_51. URL: http://dx.doi.org/10.1007/978-3-642-05089-3_51.

[21] Nancy G. Leveson. “Software Safety: Why, What, and How”. In: *ACM Comput. Surv.* 18.2 (June 1986), pp. 125–163. ISSN: 0360-0300. DOI: 10.1145/7474.7528. URL: http://doi.acm.org/10.1145/7474.7528.

[22] K. Marcinek, A.W. Luczyk, and W.A. Pleskacz. “Enhanced LEON3 core for superscalar processing”. In: *Design and Diagnostics of Electronic Circuits Systems, 2009. DDECS ’09. 12th International Symposium on*. 2009, pp. 238–241. DOI: 10.1109/DDECS.2009.5012137.

[23] M Masmano, I Ripoll, and A Crespo. “An overview of the XtratuM nanokernel”. In: *OSPERT 2005—Workshop on Operating System Platforms for Embedded Real-Time Applications*. 2005. URL: http://ecrts.eit.uni-kl.de/fileadmin/WebsitesArchiv/Workshops/OSPERT/Proceedings/OSPERT2005-Proceedings.pdf#page=31.

[24] M Masmano, I Ripoll, and A Crespo. "An overview of the XtratuM nanokernel". In: *OSPERT 2005—Workshop on Operating System Platforms for Embedded Real-Time Applications*. 2005.

[25] Miguel Masmano et al. "Xtratum: a hypervisor for safety critical embedded systems". In: *11th Real-Time Linux Workshop*. Citeseer. 2009, pp. 263–272.

[26] Miguel Masmano et al. "Xtratum for leon3: an open source hypervisor for high integrity systems". In: *Embedded and Real-Time Software and Systems* (2010).

[27] K. D. Maxwell, L. Van Wassenhove, and S. Dutta. "Software development productivity of European space, military, and industrial applications". In: *IEEE Transactions on Software Engineering* 22.10 (1996), pp. 706–718. ISSN: 0098-5589. DOI: 10.1109/32.544349.

[28] Sun Mierosystems. *The SPARC Architecture Manual (Version 8˜*. 1990. URL: http://www.gaisler.com/doc/sparcv8.pdf (visited on 05/06/2016).

[29] John Noll. "Flexible process enactment using low-fidelity models". In: *Proceedings of the International Conference on Software Engineering and Applications, Marina del Rey, USA*. 2003.

[30] John Noll and Jigar Shah. "Process state inference for support of knowledge intensivework." In: *IASTED Conf. on Software Engineering and Applications*. 2004, pp. 382–387.

[31] P. J. Prisaznuk. "ARINC 653 role in Integrated Modular Avionics (IMA)". In: *2008 IEEE/AIAA 27th Digital Avionics Systems Conference*. 2008, 1.E.5–1–1.E.5–10. DOI: 10.1109/DASC.2008.4702770.

[32] Raymond J. Richards. "Design and Verification of Microprocessor Systems for High-Assurance Applications". In: ed. by S. David Hardin. Boston, MA: Springer US, 2010. Chap. Modeling and Security Analysis of a Commercial Real-Time Operating System Kernel, pp. 301–322. ISBN: 978-1-4419-1539-9. DOI: 10.1007/978-1-4419-1539-9_10. URL: http://dx.doi.org/10.1007/978-1-4419-1539-9_10.

[33] David Sanán, Andrew Butterfield, and Mike Hinchey. "Verified Software: Theories, Tools and Experiments: 6th International Conference, VSTTE 2014, Vienna, Austria, July 17-18, 2014, Revised Selected Papers". In: ed. by Dimitra Giannakopoulou and Daniel Kroening. Cham: Springer International Publishing, 2014. Chap. Separation Kernel Verification: The Xtratum Case Study, pp. 133–149. ISBN: 978-3-319-12154-3. DOI: 10.1007/978-3-319-12154-3_9. URL: http://dx.doi.org/10.1007/978-3-319-12154-3_9.

[34] H. Shi, J. Peleska, and M. Kouvaras. "Combining methods for the analysis of a fault-tolerant system". In: *Dependable Computing, 1999. Proceedings. 1999 Pacific Rim International Symposium on*. 1999, pp. 135–142. DOI: 10.1109/PRDC.1999.816222.

[35] LEON3-FT SPARC. "V8 Processor Data Sheet and User's manual". In: *Aeroflex Gaisler* (2012). URL: http://www.gaisler.com/doc/leon3ft-rtax-ag.pdf (visited on 05/06/2016).

[36] Cary R Spitzer and Cary Spitzer. *Digital Avionics Handbook*. CRC Press, 2000.

[37] J. Windsor and K. Hjortnaes. "Time and Space Partitioning in Spacecraft Avionics". In: *Space Mission Challenges for Information Technology, 2009. SMC-IT 2009. Third IEEE International Conference on*. 2009, pp. 13–20. DOI: 10.1109/SMC-IT.2009.11.

[38] CBR Staff Writer. *SGS THOMSON GIVES UP ON T9000*. June 1997 (accessed March 2, 2016). URL: http://www.cbronline.com/news/sgs_thomson_gives_up_on_t9000.

[39] Angela Wallenburg Yannick Moy. *Tokeneer: Beyond Formal Program Verification*. URL: http://archive.adaic.com/projects/atwork/boeing.html.