"Formal methods Expert to IMA-SP Kernel Qualification — Preparation"

ESTEC Contract No. 4000112208/14/NL/GLC/

IMA-KQP Phase 4: R1: Formal Methods Expertise — Final Report

(FMEIMAKQP FR, Version 1.0)

July 21, 2016

Andrew Butterfield

Lero—The Irish Software Research Centre

Andrew.Butterfield@scss.tcd.ie

Contents

1 Change Log

Current Version: v1.0

v0.1 Initial draft based on deliverable DD6, Formal Experimentation.

v0.2 Re-structuring, putting formal models to appendices.

v0.3 Writing activity summaries.

v0.4 Writing Results and Recommendations.

v1.0 Submitted Report.

2 Introduction

This document describes results of an exploration of ways to use formal methods and techniques to assist with the qualification of Time-Space Partitioning (TSP) kernels. It is being written as part of ESTEC Contract No. 4000112208/14/NL/GLC for activity "Formal Methods Expert to IMA-SP Kernel Qualification — Preparation" (FMEIMAKQP). This activity is being run in support of ESA RFQ 3-14102/14/NL/GLC/al "IMA-SP Kernel Qualification — Preparation" (IMAKQP).

We continue here with an overview of the formal methods involvement, and then proceed to describe the formal activities associated with the four IMAKQP phases (§3–6). We then draw conclusions from the experience (§7) and propose possible future work (§8). Formal models — entire or fragmented — are included in the Appendices (§A–F), along with notes on various issues as they arose during the activity (§G).

Our exploration of the application of formal methods to kernel verification focussed on four areas. In rough chronological order these where:

1. Building a formal model of the Baseline Requirements (§A).
2. An initial exploration of the nature of the overall kernel invariant (§C.1)
3. Developing a formal model that could assist in verifying those parts of the kernel involved in trap/interrupt handling (§D).
4. Assisting ADS in their exploration of the use of Frama-C to verify XtratuM kernel code against the requirements (§E,§F).

5. A final revisit to the data invariant that characterises the valid use-cases for kernel configurations (§C.2).

Completing all of these models was well beyond the resources available, so the focus was on doing some exploration that would give good feedback on the feasibility of the various approaches.

2.1 Acknowledgments

The CSP models presented here were developed by Kevin Hennessy as part of his Masters Dissertation[1] at Trinity College Dublin.

3 Phase 1 Activities

Phase 1 of the IMAKQP activity had, as its main goal, the production of the definitive Requirements Baseline for a partitioning kernel, forming deliverable D02 [2]. The main nature of this activity was an iterative process of editing and review.

We participated fully in this activity, mainly to ensure that we had a good understanding of the nature of the proposed kernel requirements, and the various issues that were raised and addressed in order to come to an agreed baseline.

This phase ended with the Software Requirements Review and contained the Core Requirements Workshop.

3.1 Presentations

A Core Requirements Workshop was scheduled during Phase 1, and part of that was a formal methods presentation[3]. It was a description of the separation kernel formal modelling that was done in Tasks 3 and 4 of the previous MTOBSE activity. After the workshop, feedback was solicited, and a response was obtained from ADS, which among other things made useful suggestions for topics to be covered in the subsequent two workshops.

3.2 Investigation Formalisation

In parallel we also did an initial look at what formalising the requirements would involve. In effect, we needed to determine an appropriate level of abstraction to capture the essential aspects of the kernel requirements, without being swamped by too much detail. What follows here is a high-level summary of outcomes, that are presented in more detail in §A.

We started with an overview of the key kinds of entities present in the system, namely those that are physical, abstract and virtual, and the nature of the links between them. Some principles:

- The CPU executes instructions on behalf of an abstract entity.
- The term “abstract” should not be confused with “virtual”.
- A partition (abstract entity) will “see” a virtual machine while running on a physical CPU.

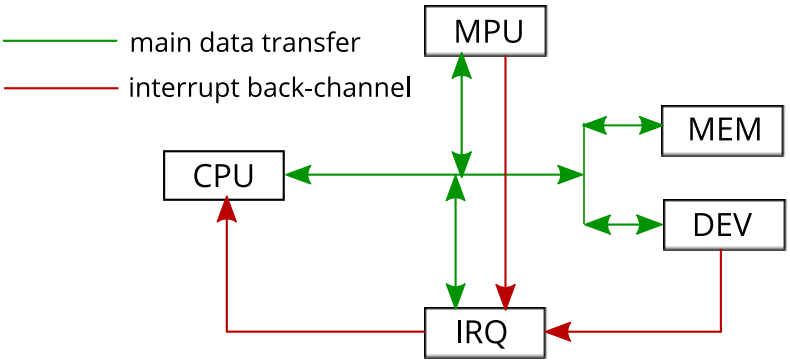

Figure 1: PK Physical Hardware and Signal links

- The kernel (another abstract entity) will "see" the physical machine while running on a physical CPU.
- Memory/MMU stores/handles data on behalf of an abstract entity.
- Memory-mapped I/O handles data on behalf of an abstract entity.
- An interrupt signal comes from physical hardware (CPU,MMU,Device) to an abstract entity.

We started by defining our foundation as a model of physical entities. The plan was to introduce abstract entities of top of this as a way of interpreting the physical model. The intention was to model the requirements as properties over both the physical and abstract entities.

3.2.1 The Physical Model

The physical model covers everything that has a physical, measurable presence, including both static parts, e.g. hardware, and dynamically changing aspects such as signal levels that change over time, or the contents of physical memory locations.

The key top-level physical hardware entities (Fig. 1) are:

CPU (or Integer Unit, in SPARC-speak), which for now includes the FPU.

MEM Memory, including both RAM and non-volatile Flash Storage

MPU Memory Protection Unit, could be MMU/MPU or otherwise.

DEV Devices, including timers and I/O

IRQ Trap/Interrupt handling hardware (part of the CPU/IU)

DATA Data. We do not consider data to be abstract, but is rather physical, and dynamic. It covers the 1s and 0s physically stored in the five (static) hardware devices above. Data resides in hardware devices at identifiable locations: specific register, memory address, etc. The corresponding abstract notion is Information.

A key assumption is that the five hardware components are all running in (true) parallel, and that DEV and any associated storage is effectively memory-mapped.

This view of the physical platform is the basis for the CSP model[4] that was developed later on [1] to explore interrupt behaviour in more detail — see §D.

3.2.2 The Abstract Model

The abstract model captures the kind of concepts that dominate the requirements: kernel, partition, schedule, FDIR, port, etc. We can consider four kinds of abstractions that we need: Entities, Events, Behaviour and Information. Abstract entities include: the Kernel, which is the object of our formal analysis; Partitions, which are the entities we manage using the kernel; Channels, permitted logical links between partitions; and Ports, which access points that allow Partitions to use Channels.

Abstract events are basically the small observable components of behaviour, and correspond to changes to the system that we consider important. For example, a key abstract concept is that of a Trap, which is the basic mechanism for both partitions and monitoring devices to signal or invoke the kernel, and are an abstract counterpart to the physical traps signalled by the IRQ device. But there are many other events, mentioned explicitly in the requirements, such as Port Creation, Message Send/Receive, or Context Save/Restore. Many of these are triggered by specific traps (physical/abstract) but need to be considered in their own right.

Abstract behaviours are basically coherent groupings of abstract Events, e.g. Context Switch, Partition Reset, or Fault Handling by FDIR.

Data is physical, but is interpreted abstractly as Information. We can classify information based on its intended (abstract) use and purpose:

Configuration used to specify to the Kernel the abstract events that are either permitted, or forbidden.

Schedule determines when each partition is allowed to run.

Messages used to model permitted information flow

FDIR Event Logs used to record detected attempts by partition to violate constraints, or failures within the kernel itself

3.2.3 Model Summary

We have discussed key concepts and components of a formal model. Initial sketches of such a model can be found in §A. The basic idea here is that the Abstract Model is intended to capture the intent of the requirements baseline, while the Physical Model describes the execution platform, and ultimately will enable a link to the real hardware platform and the software running on it. The Physical Model is itself quite an abstract view of the actual platform and code, and several levels of refinement will be required to get to code level.

3.3 Phase-1 Summary

At the end of Phase 1 the large scale of this formal modelling exercise was now becoming very clear, and it was felt that the exploration in subsequent phases should focus in on specific parts of interest. Given that ADS was going to explore the use of Frama-C[5] to verify some real kernel code, and concerns raised at meetings regarding the difficulty of doing satisfactory testing of the trap/interrupt handling, it was felt that we should focus on these aspects.

This phase ended on 27th March 2015 with the Software Requirements Review (SRR).

3.4 FMEIMAKQP Phase 1 Deliverables

DD1 MTOBSE workshop materials (Phase 1) [6]

W1 Formal Modelling Workshop (I) [3]

DD2 Input material to IMA-KP SRR document D01 "Consolidated input documentation for IMA Separation Kernel Requirements" [7]

DD3 Input material to IMA-KP SRR document D02 "IMA Separation Kernel Requirements Baseline" [8]

4 Phase 2 Activities

Phase-2 of IMAKQP was involved with defining the process for kernel qualification. Its main task was to establish the best way, for each requirement, to assess the compliance of the corresponding kernel code. It also determined the focus of the case study that would take place in Phase 3, during which a selected subset of the qualification process would be run on a trial basis.

Our focus during this phase was to explore what would be needed in order to use formal methods to verify some or all of the requirements involving traps and interrupts. This focus was adopted because it was felt that this area would be most problematic for testing, simply because of the inherent non-determinism involved.

This phase ended with the Test Readiness Review (TRR) on 23rd September 2015.

4.1 Presentations

An Extended Requirements Workshop was scheduled during Phase 2, and part of that was a second formal methods presentation[9]. The focus of this presentation was on ways of connecting the abstract models from the first workshop down to the implementation code. The technique for linking an abstract model to a more concrete one is known as "refinement", and examples illustrating were presented. The bottom-up approach using formal annotations of C was discussed, most notably with the Frama-C tool[5]. Relationships between Formal Methods and Testing were also discussed, particularly with reference to model-based testing. Finally, the issues regarding the correctness of kernel configurations was discussed.

A Qualification Methods and Techniques Workshop was also scheduled during Phase 2, including a third formal methods presentation[10]. This presentation focussed very closely on the issues that would arise in trying to use formal techniques to verify that the XtratuM kernel code satisfied the Requirements Baseline[2]. It looked at formal model analysis techniques such as process algebras for capturing top-level behaviour. It explored in detail what kind of refinement relations would be needed to get from actual requirements (PK-xxx) to relevant XtratuM code extracts. It proposed the devolvement of a formal model of the baseline, including behaviour and key data and safety invariants, and discussed the likely effort required to do formalisation.

Feedback was solicited from consortium partners after both of these workshops, but none was received.

A presentation giving a preliminary overview of the FMEIMAKQP work was given at DASIA 2015[11]. It described the results so far of Phases 1 and 2 and looked forward to the investigations to come in Phase 3.

This phase ended on 23rd September 2015 with the Test Readiness Review (TRR).

4.2 FMEIMAKQP Phase 2 Deliverables

DD1 MTOBSE workshop materials (Phase 2) [6]

W2 Formal Modelling Workshop (II), [9]

W3 Formal Modelling Workshop (III), [10]

DD4 Input material to IMA-KP TRR document D06 "Strategy and organisation for qualification of IMA Separation Kernels" [12]

DD5 Input material to IMA-KP TRR document D08 "Methods and techniques for qualification of IMA Separation Kernels" [13]

5 Phase 3 Activities

From a formal methods perspective, the main task during phase 3 was to to some exploration of formal modelling and verification of the focus areas. In effect we had determined two focus areas: interrupts, and looking at using Frama-C to verify selected hypervisor calls in the XtratuM code.

5.1 Formalising Interrupts

Traps and interrupts play a crucial role in the operation of a separation kernel. The kernel configures the target platform to ensure that appropriate traps are raised when events of interest to the kernel occur. These include alerts from the MMU regarding invalid memory accesses, or expected I/O device interrupts.

A partition's only way to invoke kernel services such as IPC is by using a trap instruction. Kernel correctness depends on it ensuring that the right settings in hardware are in place at the right time. Add to this the fact that the timing of interrupts is effectively non-deterministic, originating often from hardware running in parallel with the CPU, then we can see that there is quite a verification challenge, regardless of which method is being used.

Some aspects of the trap infrastructure were quite easy to formalise, for example the terminology used in the requirements baseline[2]

$$\begin{aligned} TRAP &= SWTRAP \mid EXCPT \mid HWINT \\ EXCPT &= NOMINAL \mid ERROR \end{aligned}$$

A trap is virtualized if it leads to a partition-provided handler being invoked. The *variety* of traps, both physical and virtual requires careful modelling. The key issue is to capture how interrupts are prioritised, and enabled or disabled. Again, this is is straightforward to formalise, and to validate. For example, Fig.2 contains a formalisation of the description in the Leon3FT manual [14, §9.2.1,p77] regarding interrupt priorities. These are all quite simple models of what is either structural information or straightforward functional behaviour. Most of this can be treated simply and abstractly.

The real problem with reasoning about interrupts, either formally or informally, is not the fairly static descriptions of interrupt numbers, priorities and masking, but rather the *dynamic* behaviour of a system in which they can arise at unpredictable times.

The real challenge arises because we have a number of hardware entities (CPU, MMU, MEM, DEV, IRQ) all running in parallel (Fig. 1,p6). The key feature here is both CPU and DEV are effectively independent, but the DEV behaviour looks completely non-deterministic from the perspective of the CPU.

The CPU's responsibility (when running kernel code) is to ensure that MMU, DEV and IRQ are configured so that when it reverts back to partition code, then IRQ properly handles both anything flagged, correctly, by the MMU, and anything, no matter how unexpected,

$$
\begin{array}{rcl}
Irq, Prio \in IRQ & = & \mathbb{P}\{1,\ldots,15\} \\
cpu \in CPUId & & \text{—CPU Identifier} \\
force \in FORCE & = & CPUId \to IRQ \\
mask \in MASK & = & CPUId \to \{1,\ldots,14\} \\
raise & : & CPUId \to IRQ \to \{0,\ldots,15\} \\
raise(cpu)(Irq) & \widehat{=} & pencode((Irq \cup force(cpu)) \cap (mask(cpu) \cup \{15\})) \\
pencode & : & IRQ \to \{0,\ldots,15\} \\
pencode(Irq) & \widehat{=} & \textbf{let } Hi = Irq \cap Prio \\
 & & \textbf{in if } Hi \neq \emptyset \\
 & & \textbf{then } encode(Hi) \\
 & & \textbf{else } encode(Irq) \\
encode & : & IRQ \to \{0,\ldots,15\} \\
encode(\emptyset) & \widehat{=} & 0 \\
encode(Irqs) & \widehat{=} & \max Irqs \\
 & & \\
ack & : & IRQ \to (IRQ \times FORCE) \to (IRQ \times FORCE) \\
ack(irq)(Irqs, Force) & & \\
 & \widehat{=} & \textbf{if } irq \in Force \\
 & & \textbf{then } (Irqs, Force \setminus \{irq\}) \\
 & & \textbf{else } (Irqs \setminus \{irq\}, Force) \\
 & & \\
reset & : & (IRQ \times FORCE \times MASK) \\
reset & \widehat{=} & (\_, \emptyset, \_)
\end{array}
$$

Figure 2: Formalising Interrupt Priorities [14, §9.2.1,p77]

that comes from DEV. When IRQ raises a trap with the CPU, then the handlers, which are considered as kernel code, must do the correct thing.

We decided to use the modelling language called Communicating Sequential Processes (CSP)[4] to model (i) the physical hardware as shown in Fig.1; and (ii) an abstract view of the running kernel and partition software; and (iii) abstract models of the baseline requirements. The plan was to validate the model by showing that the interaction of these two models satisfies the baseline requirements. In CSP this is typically achieved by putting the various models in parallel, specifying how they must synchronise, and then verifying that the combined model has key properties. A common approach is either to show that one model refines another (this can be used to verify both liveness and safety properties), while another is to setup the combined model so that if the property is true then the model is deadlock-free, while falsity is indicated by a deadlock with the path that leads to the deadlock being a counter-example.

We did not intend to prove any such properties, but instead made use of the FDR3 refinement-checking tool[15, 16] for doing automated exhaustive testing of the models.

This investigation was largely done by a Masters student at TCD, Kevin Hennessy, and formed part of his dissertation[1].

The approach adopted was to start with the hardware model, and try to find the simplest model that has all the relevant features. Relevance here means that our main concern is being able to capture all the pertinent interrupt handling behaviour. A real problem that had to be addressed was keeping the size of the model's state-space from suffering a combinatorial explosion that would have forced the FDR3 tool to run out of memory. The key was to have the ability to have different models of each component (CPU,IRQ,...) tailored to the specific scenario under investigation. For example, keeping track of the contents of memory could prove to be very expensive in terms of state-space, so we had memory models that did not do this if it was not relevant (e.g. if we wanted to model the ability of the MMU to allow (prevent) wanted (forbidden) memory accesses, then the address and access mode are important, but not the data).

For example, we consider some examples models of MEM and MMU. First, we define types and channels to abstract memory accesses:

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

The memory bus activity starts with a global clock tick, and does one of three alternatives based on external events.

```
--m is the memory contents
MEM_BUS(m)
=   tick -> ( MEM_BUS_READ(m)
            [] MEM_BUS_WRITE(m)
            [] MEM_BUS(m) )
```

If the CPU initiates a memory read, then the memory waits for clearance from the MMU. If given, it then looks up its contents and performs a read data transfer. If the MMU signals an invalid access, then MEM does nothing.

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

Memory writes get treated in much the same way, except that a write data transfer occurs, if permitted by the MMU.

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

In all the above cases, once memory has behaved as instructed it recursively goes back to the first behaviour, waiting for the next clock tick.

The MMU needs to be able raise a trap with IRQ, and we model writes to it from the CPU to change its configuration as specific events, distinguished from general memory traffic.

```
channel raise : InTag   --synced with IRQ
channel mmuBlock, mmuUnBlock : Mem_Region    --synced with CPU
```

The MMU waits for the global clock, and them exhibits one of three behaviours, based on external events.

```
-- blocked : set of blocked memory accesses
MMU(blocked)
=   tick -> ( TRANSFER(blocked)
           [] BLOCK_MEM_REGION(blocked)
           [] UNBLOCK_MEM_REGION(blocked)
           [] MMU(blocked) )
```

If the MMU sees a bus transfer, initiated by the CPU, it checks to see if it is blocked. If not, it signals that the memory operation may proceed. otherwise it signals a bad memory access, and raises an appropriate trap.

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

                    else ( mmuOK -> MMU(blocked) )
                   )
```

Events like `bus?dir?addr` and `raise!memfault` correspond to real signals in the platform being modelled. Events like `mmuOK` and `badaccess` are part of the modelling abstraction. They either act as markers so we can observe what is happening, or, as here, they also serve to signal to something like MEM if the relevant bus operation is going to continue, or abort.

Another possibility for the MMU is that the CPU is changing its `blocked` set.

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

The issue of when these are legal (the CPU is in supervisor mode) is handled by the CPU model—both the MEM and MMU essentially do what they are told.

Initial thoughts were to model kernel and partition software as code stored in memory, with its address being used, in conjunction with configuration data, to determine if it was kernel or partition code. However this kind of model immediately leads to a very large combinatorial state explosion. Instead a technique was developed, were each "thread" (kernel or partition), was modeled as a process that supplied abstract instructions on demand from the CPU component. This allowed us to design specific small scenarios in which we could investigate specific behaviours of interest.

In particular, it allows us to model a baseline requirement as a relatively simple scenario, and ideally obtain a faithful model that is tractable for model-checking. Basically a requirement model is one that runs in parallel with the hardware and software models, and which either signals success with a special event, or deadlocks the system if the requitement is violated.

We give a brief example of how a test might be put together. This test fails if partition `2` manages to write to address `a0`. It succeeds if the partition tries, but the kernel has correctly configured the MMU to raise an exception and to abort the write. We put a checker process in parallel with the "system" synchronising on relevant events

```
INTERNAL_SYSTEM5
 = INTERNAL_SYSTEM4
   [| {| tick, fe.partition2.writeInst.jDC.a1.d0, write.d0, mmuOK |} |]
   CHECKER
```

Event `fe.partition2.writeInst.jDC.a1.d0` denotes partition 2 attempting such a write, and event `write.d0` will occur immediately afterwards if it succeeds.

The checker process continually performs clock ticks but also keeps an eye out for a write attempt followed by a write success, in which case it emits a loud test failure event, and then stops performing any events (i.e. deadlocks). In particular it will refuse to perform any further tick events so deadlocking the whole system.

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

The partition 2 event will be omitted by the CODE process, which also is put in parallel

```
INTERNAL_SYSTEM6
 = INTERNAL_SYSTEM5 [| {| fe, SCENARIO_COMPLETED |} |] CODE
```

All of this is in parallel with the clock to produce the full system, which we assert should be deadlock-free.

```
SYSTEM = INTERNAL_SYSTEM6 [| {| tick |} |] CLOCK
assert SYSTEM :[deadlock free [F]]
```

The current state of the CSP modelling is that of a proof-of-concept. There is still a need to build and check models of all the relevant requirements. It also needs some clean-up work, and more careful validation against the Sparc/LEON documentation.

The benefit of this modelling approach is that each scenario showing the compliance of a a fragment of the kernel behaviour to a relevant requirement, also serves to give a partial specification for the corresponding C code.

5.2 Exploring Hypercalls

All our exploration of actual kernel code was based on the XtratuM code version v3.3-1.3 that we had obtained for the prior MTOBSE project. Part of this was done by ADS, investigating the use of Frama-C to verify if some hypercall implementations satisfied the requirements. That work mainly looked at some hypercall code in isolation and explored what kinds of annotations made sense. They chose to look at the following two hypercalls:

SwitchSchedPlanSys This is a request to change schedule plans. This call will only succeed if the calling partition is authorised to make this call.

SuspendPartitionSys This is a request to suspend an identified partition. This always succeeds if a partition requests its own suspension (a.k.a. "yielding") but only allows authorised partitions to suspend others.

A detailed report of their investigations can be found in IMAKQP deliverable D23[17], to which we also contributed.

Here we discuss some additional exploration we did that tried to look at how we would go about creating a formal "evidence chain" that would connect the low-level Frama-C annotations up to some formal model of the relevant requirements. We identified the following requirements as relevant to this study focussing on the traps and their handling: PK-230, 99, 58, 59, 67, 60, 61, 69, and 186.

The basic idea was to try and establish how these hypercalls were invoked, so as to establish the precise context in which they ran. This would help us identify what pre-conditions would be guaranteed by their callers, which we could then assume as given, in order to prove their behaviour was correct.

What emerged was that there are two distinct calling routes, one used by partitions, and a different one employed by Health Monitoring (HM). For a partition, it issues a trap instruction, which then jumps through the corresponding trap table entry. The table entries are setup by the kernel and all lead to kernel handler code. The jump puts the processor into supervisor mode. However we were unable to identify how a hypercall in the C code of a partition:

```
1 retValue= XM_suspend_partition(P2);
```

turns into a trap instruction and trap number. All we could find in the sources regarding the above function was:

```
1 extern __stdcall xm_s32_t XM_suspend_partition(xm_u32_t partition_id);
```

We did find assembler code that setup the relevant trap table entries, but no code-based link between those and the above.

Our attempt to establish a formal chain was not successful, because our attempt to identify the call chains from partitions to hypercalls took a lot of code searching, and even then

resulted in gaps we could not fill. This brings home the point, also made by ADS in their Frama-C investigations, that a high degree of familiarity with the code and the Makefiles that compile the code is needed in order to navigate it to identify how it all connects.

In fact a major problem with the XtratuM sources *as given*, is the complete lack of any (obvious) high-level overview or notion of kernel design. The other problem is that the sources are designed to be a single repository for all code that ports XtratuM to a wide range of platforms. It is very hard to establish precisely which bits of the code will end up being part of an actual kernel running on the IMAKQP target [platform, namely the LEON2+MPU/LEON3 Sparcv8 architectures.

It really does require a full-time commitment by one of more individuals to do this task effectively, in conjunction with access to someone who is very familiar with the C code and its architecture.

5.3 Kernel Invariant Revisited

Armed with considerably more familiarity with the requirements for the proposed kernel, its target platform, and, to a lesser degree, with the XtratuM code, we revisited the Requirements Baseline and did a systematic trawl through trying to identify key data invariants at a high level of abstraction.

As a result, the following list of such invariant properties emerged, presented here in the order in which the need for them was encountered. Not all of them have actually been formalised as of yet. Also, some of the "invariants" found are not static properties of data, but are in fact dynamic or behavioural invariants (i.e., an allowed memory access always succeeds). We indicate for each invariant if it is static or dynamic in character.

1. [Static] Partition start addresses must be in RAM or random-access NVR
2. [Static] Every partition has an initial context, which zeros registers, and resets virtual traps and pending interrupts.
3. [Static] There exists at least one partition authorised to start other partitions.
4. [Static] No two partitions should share the same memory addresses (Possible Exception: shared messaging space? e.g. "zero copy"?).
5. [Dynamic] A memory access by a partition to its authorised memory always succeeds.
6. [Dynamic] A memory access by a partition to memory for which it has no authorisation always fails.
7. [Dynamic] The memory access permissions for a partition are time-invariant.
8. [Static] Basically every partition has a sequence of timeslots, usually defined as by start/stop time offsets within a major time frame.
9. [Static] Partition execution timeslots never overlap.
10. [Static] Every scheduler plan ensures that a slot is associated with exactly one partition.
11. [Dynamic] Partitions that have yielded have their virtual interrupts masked.
12. [Static] Each port is owned by one partition: $Port \rightarrow Partition$.
13. [Static] Port names are unique within a partition.

14. [Static] For every source (destination) port in one partition there must exist at least one corresponding destination (source) port in another (or same!) partition.

15. [Static] For any port there is exactly one partition in which there is a corresponding source port.

16. [Static] A Sampling channel has one Sample source port in a partition and one or more Sample destination ports in one or more partitions.

17. [Static] A Queuing channel has one Queuing source port of non-zero size in a partition and one Queuing destination port of the same size in a partition.

18. [Static] Different physical interrupts can be routed to one partition, but one physical interrupt cannot be routed to different partitions.

19. [Dynamic] Kernel configuration should ensure that all the above traps and errors are passed to HM.

20. [Dynamic] Partition interrupts are masked when not it is running.

In essence we have a long list of various properties that must hold for a valid configuration. What needs to be done is ensure that all the ad-hoc formal notation used in the appendices here (§A,§C,§D), is well-defined, and all talks about the same concepts. Then the final grand invariant is simply a giant conjunction (logical-and) of all the small invariant pieces.

However, we also had some questions regarding how ports in different partitions are linked to form channels. It was not clear precisely how ports from different partition were linked together. We decided to look at some of the example Xtratum XML config files to see how they did it. In essence we found that XtratuM allows the definition of "channels", which are basically a list of pairs consisting of partition identifiers twinned with a name of a port declared as being associated with that partition. This is not described in the requirements baseline, as it is considered a configuration implementation detail. While this may be so, we just observe that almost any way of specifying how partition ports are linked together will need to group them, each of these groupings effectively being a channel or link, even if not named. We have a complex invariant here that needs to ensure that the specification of ports in partitions are consistent with the groupings that are specified somehow to produce channels or links.

5.4 Presentations

There were no presentations given during Phase 3.

5.5 FMEIMAKQP Phase 3 Deliverables

DD6 Input material to IMA-KP QR document D11 "Evaluation of methods and techniques for qualification of IMA Separation Kernels" [18]

6 Phase 4 Activities

Phase 4 was mainly concerned with final reporting on the activity as a whole (including this report).

6.1 Presentations

A talk summarising the results of our experience in looking at formal aspects of kernel qualification was presented at DASIA16[19]. It focussed on the target area of interrupts and hypercalls.

A 20 minute presentation on FMEIMAKPQ was also given, as part of the overall IMAKQP presentation[20], at the TEC-SW Final Presentation Days at ESTEC on June 9th 2016., as part of IMAKQP deliverable D16.

6.2 FMEIMAKQP Phase 4 Deliverables

DD7 Input material to IMA-KP QR document D14 "IMA Kernel Qualification Final Report" [21]

DD8 Contract Closure Summary Report

R1 R1: Formal Methods Expertise — Final Report [22] (this report).

7 FMEIMAKQP Results

Here we give a top-level summary of the main results of this activity.

Early work looked at what would be involved in building a formal model of the baseline requirements. Various fragments of formal models of different aspects were produced, as can be seen in Appendix A. However given the focus of IMAKQP on qualifying actual kernel code, the emphasis switched away from this top-level formalisation, towards a stronger focus on code verification. It was decided to limit the scope of the formal investigation to requirements involving interrupt handling. This was because the behaviour, timing and unpredictability of interrupts meant testing was expected to be be difficult.

For the later investigations it was decided to perform two case studies studies during Phase 3. TCD would use process algebra to model essential concurrency in the system at a high-level. Airbus DS would explore the use of the Frama-C tool to verify selected XtratuM hypercalls, down at the level of actual code.

7.1 High-Level CSP Model

In a single-core system, the CPU is time-shared between the hypervisor and partition code with no (apparent) parallelism, with flow-control change managed via traps. However, even in the single-core case we have an essentially concurrent system, in particular there are interrupts, most notably from timers and I/O devices which are running in parallel to the CPU. This concurrency introduces non-determinism into the mix and the kernel's correctness depends on it being able to handle it properly. As this non-determinism was a potential source of difficulty for testing approaches, it was decided to explore a high-level abstract model of the system that would capture this concurrent and non-deterministic behaviour.

The plan was to use the Communicating Sequential Processes (CSP) notation[4] to model this behaviour and to use the FDR3 tool[15] to do analysis. This tool uses model-checking techniques to test models of concurrent systems for properties such a deadlock or livelock, or to establish if an abstract model is refined by a more concrete one.

Components (Physical platform, Software, Requirement models) run in parallel with common events on which they must agree. The platform model should be deadlock-free. The software model should also run continuously, containing processes that model both the kernel and the

partition software. The requirement models are designed to deadlock if they observe events that are inconsistent with themselves. All of this is put in parallel in such a way that requires them to all agree on events they care about. The kernel's ability to satisfy a requirement can be modelled by providing partition code that tries to violate that requirement, and seeing if the resulting system deadlocks.

This CSP model is quite a long way from the C code implementation. However this does not prevent it from being useful. First, it helps us to understand the relevant interactions between the relevant fragments of kernel and partition code. In particular it can expose the degree and nature of non-deterministic interleaving of events of interest. The FDR3 tool can generate graphs illustrating this non-deterministic interleaving of events. This will help in guiding the determination of the appropriate pre- and post-conditions that need to be associated (at the Frama-C level) with the corresponding parts of the C code. In effect, the validated CSP models will give a high-level "shape" to the specifications required at a lower level for code verification. The models can also assist in ensuring full coverage by both formal and test-based verification techniques, because we can enumerate all the different execution paths through the system.

CSP models very quickly get too large for the tools to handle. While FDR3 supports model-checking in the cloud to help mitigate this problem, it is still easy for the state-space size to quickly go beyond any amount of available resources, even at a global scale. Very careful abstraction is required to minimise state size, while retaining modelling accuracy.

7.2 Low-level Frama-C Annotations

Airbus Defense and Space conducted a review of formal code verification based on the use of the Frama-C toolset[5]. The plan was to take a few hypercalls and explore what was involved in producing and checking some formal annotations of relevant parts of the XtratuM code.

Their experiences were reported in D23[17]. In effect they were able to annotate some of the code associated with the two chosen hypercalls, and were able to successfully use the toolset to verify some of the resulting proof obligations. However, because of the difficulty in navigating the XtratuM source without help from the developers, their formal annotations were based on an analysis of the behaviour of the C code, particularly the extensive checks it performed, rather than on properties derived from the requirements. In addition, the XtratuM code had to be edited slightly, to make it compatible with the kind of C code that the tool could handle.

Frama-C supplies both automatic checking of annotations, as well as interactive approaches. The automated checking was used in experiments, and successfully verified a significant proportion of the conditions. For those conditions that proved too complex for automatic verification, the next step would have been to invoke and use the interactive theorem-proving facilities. ADS did not do this due both to lack of available time for this activity, and lack of technical expertise. Unfortunately, while state-of-the-art interactive theorem provers can be very powerful (even astonishingly so!), there is a very steep learning curve in order for the user to be really proficient in their use. Also, a bit like piloting a commercial aircraft, it is necessary to use these tools continuously in order to remain proficient.

Unexpected Challenge

A number of baseline requirements were selected in advance, during Phase 2, as possible candidates for formal verification. The main criterion for their selection was that they looked like they might be hard to test. In particular the behaviours associated with tests

to check requirements regarding interrupts looked like they would involve non-deterministic behaviour with unpredictable timing.

One result of Phase 3 however, was that a requirement not explicitly about interrupts, namely PK-9, proved hard to test. It is the requirement that the kernel properly save and restore partition state at a context switch. This requirement was flagged as a possible candidate for formal verification early on, but then became sidelined by the focus on interrupt-specific requirements. Ironically, the difficulty with testing PK-9 compliance arises because of the difficulty in precisely timing the observation of the state immediately before and after the context switch. Direct observation was not possible, and so breakpoints using traps had to be used, which complicated the checking of before- and after-states.

Invariants

The notion of a data-invariant is key to verifying the correct behaviour of the kernel. Such invariants arise at both high and low-levels. At high-levels for the PK, data invariants are concerned with ensuring that the configurations are correct. The kind of properties we would consider at this level include:

- Every destination port has an associated source port, of the same type.
- No two ordinary partitions share physical memory
- No two partitions have overlapping timeslots in the MAF

At the low-level, in the kernel implementation, much of the configuration data is encoded as C datatypes. The high-level data invariants then map down to corresponding low-level ones on the C datatypes. A key issue here is that the proof of correctness of kernel code will depend crucially on those low-level invariants being satisfied. This is primarily because most of the kernel code will, for efficiency's sake, assume that these invariants hold. In general it is very expensive for low-level code to check such invariants for every operation. Note, this does not prevent low-level invariant checking being used as part of an system integrity check as part of a boot/re-boot/sever-fault recovery process.

We have catalogued some of the high-level invariants, and formalised a few. We have not really addressed any low-level ones. Any top-level formalisation of the requirements baseline itself would require formalising such data-invariants as part of the description of the behaviour both permitted and forbidden by the requirements.

Results

In brief, FEMIMAKQP was a small consultative and exploratory activity, small at least relative to the effort to do the job completely. The key observations we can make from this work include:

- High-level modelling helps to define an overall architecture for the verification task, as well as support top-level consistency checking. The architecture so provided is a combination of structural decomposition of the verification task into fairly independent sub-problems, along with a characterising of the kind of properties that will be needed at various points in the decomposition. For example some properties will have concurrency as an essential aspect, while others will simply involve sequential behaviour.
- A partitioning kernel is essentially a concurrent system, even if the implementation technology is a single-threaded, single-core processor. This arises conceptually, as

the requirements talk about the kernel acting as an on-going overseer of partition activity, intervening when necessary to provide services, or prevent illegal behaviour. The concurrency is also physical, in that the kernel's guardianship is implemented by having hardware running in parallel such as the MMU or Timer that interrupt the current partition at the appropriate time. We have explored the use of CSP as a modelling tool to help us expose essential concurrency, while avoiding that which is irrelevant.

- The case-study by ADS with Frama-C has shown that low level annotations can indeed work, bit that the connection from the code-level to the requirements level is non-trivial. In effect we need a semantic bridge between C data and kernel concepts. This requires the modelling of the kernel Data-Invariants at both the Requirements and Code levels.
- Any successful attempt to verify kernel C code, either in whole or in part, will need access to a curated version of the source code. Without such guidance from the kernel developers (architect, designer, coder) the task is too large, and many proof attempts will fail due to inaccurate assumptions made by verifiers about the context in which a code fragment is actually executing. We really need the "domain experts" involved!

8 FMEIMAKQP Recommendations

We split recommendations into two distinct parts. The first deals with possible future activities in the area of formal methods for kernel verification. The second looks at how formal techniques might be fruitfully employed in other areas of on-board software modelling and verification. We finish with a brief discussion of various modalities by which this work could be done.

8.1 Kernel-related

8.1.1 Formal Model of Requirements Baseline

This would be a comprehensive formalisation of the requirements. In effect it would describe a relationship between a valid kernel configuration and the permissible behaviours associated with both the kernel and all the partitions defined in the configuration. The key concept would be the notion of the running partition attempting to perform various actions, with the kernel permitting or blocking these according to configuration. It would require the development of a full configuration data invariant, as this would be the basis for defining "valid" configurations. The primary value of this model at the present time, given that we are looking at qualification to level B, is that it would provide a way to check the requirements baseline for internal consistently. For level A qualification such a model would almost certainly become mandatory, especially if Common Criteria security standards (e.g. SKPP) might apply.

8.1.2 Formal Model of Data Invariant

We have data invariants at various levels. At the top is the configuration invariant, expressed in terms of requirements-level concepts. This would emerge from the activity described above. As a standalone activity, its utility would be that it could be used as a basis for verifying kernel configuration tools.

8.1.3 Verifying PK-9

Given the difficulties encountered when testing PK-9, it might make sense to explore a fully formal verification of the relevant code. In effect this would require a deep-dive, from a formalisation of PK-9, down to the relevant annotation of the code using Frama-C. In order to formally link these two levels we would require the relevant high and low invariants to be determined, as well as the formal refinement link between these two levels. In addition we would need to determine how best to formalise the use of assembler, and the relevant hardware/platform behaviour. Apart from knowing if PK-9 has been properly implemented, the real value of this is that it would smoke out a lot of the issues that would be involved in a top-to-toe verification of the whole kernel.

8.1.4 Completion of the CSP model

The CSP model developed by Kevin Hennessy[1] is very much a proof-of-concept. The completion of it might be an worthwil exercise. At the very least we could investigate if the use of CSP like this is a reasonable way to formalise all/most of the baseline. CSP's main mode of operation is to have processes in parallel, with the possibility that processes can veto the behaviour of others. This look like a good fit to the idea presented above of partitions attempting actions while kernels accept /reject the attempt. This activity could be extended with an exploration of how any such CSP requirement models could be used to produced a formal specification that is closer to the kernel code functionality.

8.2 General On-Board Software

Top-to-toe formal verification from requirements/specifications down to code is feasible, but it is difficult and expensive. However, one area were formal models can deliver good value at relatively low cost, is in the abstract, mathematical modelling of high-level aspects of systems, such as requirements, architecture and high-level system models. There is a lot of such high-level modelling going on in the onboard software domain, and some of this looks like good candidates for formalisation. Of particular interest are those activities were the models and architectures are still being developed, with key design decisions still to be made. It is at these early-stage high-level phases that formalisation is at its most effective, helping to detect systemic high-level errors.

Possible candidate areas for this are in the current SAVOIR activities, covering Reference Architectures; Specifications; Data Models; Interface Standardisation; and Protocols.

8.2.1 Formal Software Engineering

Many successful uses of formal methods on an industrial scale (e.g. NICTA's sel4 [23]) have occurred in a closed environment. How can ESTEC encourage the development of a formal software engineering framework that makes this feasible in an open environment? This would be very much a long-term activity. An initial study might look successful activities such as those of NICTA just mentioned, or the work done by ClearSy to formally verify train and platform-door control systems[24][25] that are currently running live in metro systems worldwide. These would be contrasted with attempts made in an open environment, in order to try establish what really are the drivers for success in the closed approach. The hope would be that some coherent plan might emerge as to how these drivers could be adapted for an open environment. Of all the proposals listed here, this is the most "blue-sky"!

8.3 Activity Modalities

The most obvious modality for taking any of the above proposals forward is as an ESTEC funded-activity, which would typically require the employment of someone at post-doctoral level for 12-18 months, and involve some other parties to some extent if we are going to work with kernel code, or emerging high-level software models. The key advantage of this modality is that with someone involved full-time, and with this level of experience/expertise, we are likely to get good results on which we can build, at an appropriate Technology Readiness Level (TRL).

Other modalities that can used are based on getting students to take on parts of these investigations as undergraduate final-year projects, or at masters dissertation level. Some assistance of the latter kind was used for this activity. This modality has the advantage of requiring a much lower commitment of resources, but carries a much higher risk of having a relatively poor outcome — the students will be less experienced, and will not be able to devote all their time to this kind of activity. However one advantage of this kind of activity, and why I propose to engage in it some degree in the sequel, is that it will keep me "in the loop", so to speak, and ensure I stay reasonably current with developments in this area.

A third modality is getting a PhD student to do a more in-depth exploration. This usually results in good outcomes, as PhD students are significantly more experienced/mature than undergraduate or masters students. However, it would be a long-term activity (3 years minimum, 4 years typical), and likely lead to good results, but at a low TRL.

The remainder of this document are appendices which contain (mainly rough) material, some of which has already appeared as part of deliverable DD6. They are included here to keep this report self-contained.

A Formalising Requirements

Initially we did an initial exploration of what concepts and components might be needed to from a full formal model of the Requirements Baseline (RB). It was always an aspiration to construct such a model, and although it has not emerged from this activity, it is still something that might be worth pursuing. What we present here is a collection of thoughts and ideas, followed by a summary, in which we discuss the resources that would be needed to complete the task.

We started with a high-level take on a starting formal model, starting with a overview of the key kinds of entities present in the system, namely those that are physical, abstract and virtual, and the nature of the links between them. Some principles:

- The CPU executes instructions on behalf of an abstract entity.
- The term "abstract" should not be confused with "virtual".
- A partition (abstract entity) will "see" a virtual machine while running on a physical CPU.
- The kernel (another abstract entity) will "see" the physical machine while running on a physical CPU.
- Memory/MMU stores/handles data on behalf of an abstract entity.
- Memory-mapped I/O handles data on behalf of an abstract entity.
- An interrupt signal comes from physical hardware (CPU,MMU,Device) to an abstract entity.

We started by defining our foundation as a model of physical entities. The plan was to introduce abstract entities of top of this as a way of interpreting the physical model. The intention was to model the requirements as properties over both the physical and abstract entities.

A.1 The Physical Model

The physical model covers everything that has a physical, measurable presence, including both static parts, e.g. hardware, and dynamically changing aspects such as signal levels that change over time, or the contents of physical memory locations.

A.1.1 Physical Hardware

The key top-level physical hardware entities are:

CPU (or Integer Unit, in SPARC-speak), which for now includes the FPU.

MEM Memory, including both RAM and non-volatile Flash Storage

MPU Memory Protection Unit, could be MMU/MPU or otherwise.

DEV Devices, including timers and I/O

IRQ Trap/Interrupt handling hardware (part of the CPU/IU)

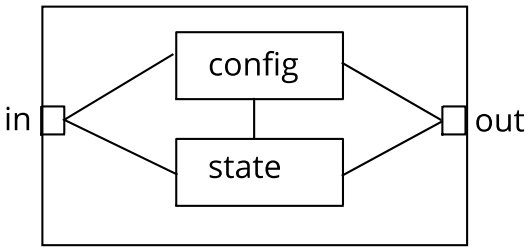

Figure 3: PK Physical Hardware and Event links

DATA Data. We do not consider data to be abstract, but is rather physical, and dynamic. It covers the 1s and 0s physically stored in the five (static) hardware devices above. Data resides in hardware devices at identifiable locations: specific register, memory address, etc. The corresponding abstract notion is Information.

A key assumption is that the five hardware components are all running in (true) parallel, and that DEV and any associated storage is effectively memory-mapped.

We can view a hardware element as a white box, with one generic input, one generic output and two internal DATA components: *config* and *state* (See Fig.3). The config data determines how the hardware responds to generic input events, which typically results in the state data being changed, and some generic output produced. Some input events are interpreted differently, as being requests to change the configuration. Generally the config data changes infrequently, and guides behaviour, while the state data is being continually updated and modified.

Out hardware elements config and state are as follows:

CPU config: user/supervisor mode; state: all registers

MEM config: none; state: everything

MPU config: defines permitted MEM accesses; state: none

DEV config: I/O settings; state: I/O buffer contents

IRQ config: trap priorities and masking; state: pending request queue.

A.1.2 Physical Signals

We have a notion of physical signals, which consist of a collection of physical communication actions between the hardware components:

Fetch CPU issues address data to MEM, and reads back instruction data (if not interrupted by the MPU).

Read CPU issues address data to MEM/DEV, and reads back data (if not interrupted by the MPU).

Write CPU issues address data to MEM/DEV, and writes over data (if not interrupted by the MPU).

MemFault MPU detects invalid address data, and sends interrupt to IRQ.

DevIRQ DEV (asynchronously/non-deterministically) decides to send interrupt to IRQ.

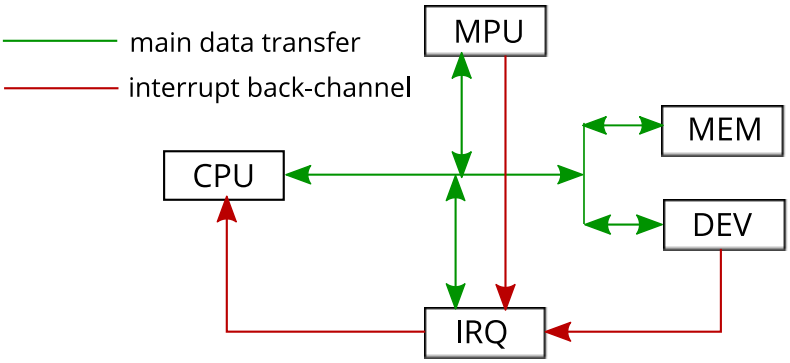

Figure 4: PK Physical Hardware and Signal links

Interrupt IRQ raises interrupt to CPU

Exception CPU raises exception to self

Here we view both CPU and DEV as autonomous in that they can "decide" what do, whereas MEM, MPU and IRQ are reactive, responding to signals from outside (See Figure 4).
This view of the physical platform is the basis for the CSP model[4] being developed to explore interrupt behaviour in more detail — see §D.

A.1.3 Physical Data

Data is physical, as already alluded to, but we can further classify data, based on its physical location in hardware:

CPU Data in CPU registers and status bits

Memory Data in all of MEM

Device Data in device control registers and data buffers

MPU Data in MPU control registers and tables

Interrupt Data in IRQ tables and interrupt queues

A key subset of all of this is what we shall call Configuration Data, most notable being the MPU and Interrupt Data.

A.1.4 Physical Timing

Most of the hardware works in tandem with some form of system clock. In particular this guides the timing of CPU operations. We considered modelling this as a two-phase process demarcated by clock-ticks. Something along these lines are emerging from the experiment in formalising traps and interrupts (§D)

A.1.5 Physical Operations

This includes coherent sequences of signals leading to some form of state (or configuration) change:

Save saving the CPU registers

Restore restoring CPU registers

The corresponding abstract notion is that of Behaviour.

A.2 The Abstract Model

The abstract model captures the kind of concepts that dominate the requirements: kernel, partition, schedule, FDIR, port, etc ...

A.2.1 Abstract Entities

Kernel the object of our analysis

Partition the entities we manage

Channel permitted links between partitions

Ports access points for Channels

Time

A.2.2 Abstract Events

Traps abstract counterpart to a physical interrupt.

Port Creation

Message Send

Message Receive

Context Switch (a behaviour?)

A.2.3 Abstract Information

Data is physical, but is interpreted abstractly as Information. We can classify information based on its intended (abstract) use and purpose:

Configuration used to specify to the Kernel the abstract events that are either permitted, or forbidden.

Messages used to model permitted information flow

FDIR Event Logs used to record detected attempts by partition to violate constraints, or failures within the kernel itself

A.2.4 Abstract Behaviour

Coherent groupings of abstract Events, e.g.

Context Switch (an event?)

Partition Reset

Fault Handling by FDIR

A.3 Model Summary

We have presented a number of lists that discuss key concepts and components of a formal model. However, we have not shown any formal notation. We now turn to look at what initial formal model emerged from a preliminary scan of the requirements.

sub

B Partitioning Kernel Model

This was a first cut at a top-level PK model.

General Principles: we typically model an instance of a class entities in the first instance as a finite map from an entity identifier to a tuple/record of its attributes:

$$Id \to AttrRec$$

Such a finite map is best vizualised as a table with two columns, the first containing values of type Id, while the second contains corresponding values of type $AttrRec$. A key feature of the table is that the values of type Id are keys, and each such value occurs at most once in the table. Entity classes that have only a single member (e.g the PK itself) are modelled as a tuple/record of their attributes.

We use [RB §M.N, pX] to refer to Section M.N, page X of *IMA-KQP D02: Partitioning Requirements Baseline, Issue 1.2* The Principles Overview spans [RB §3, p19–32], while the single-core requirements themselves span [RB §4, p33–74].

Here we just build a catalogue of types that capture the content of key kernel requirement concepts

B.1 Base Types and Parameters

$p_{id} \in PartId$		—Partition Identifier	
$m_{cpu} \in CPUMode$	$=$	$Spvr \mid User$	[RB §3.1, p19]
$sp_{id} \in SchedPlanId$		—Scheduling Plan Identifier	
$VCPUId$		—Virtual CPU Identifier	
$Duration$	$:$	$\mathbb{N}$	
$timestep$	$:$	mS	[RB §3.1, p20]
$Start$	$:$	$\mathbb{N}$	
$tr \in Trap$	$=$	$SWTrap$	[RB §3.4.1, p25]
	$\mid$	$Exception$	[RB §3.4.1, p25]
	$\mid$	$Interrupt$	[RB §3.4.1, p25]
	$\mid$	$Extended$	[RB §3.4.2, p26]
$he \in HMEvent$	$=$	$Scope$	[RB §3.5, p27]
	$\times$	$PartId$	[RB §3.5, p27]
	$\times$	$Cause$	[RB §3.5, p27]
$Scope$	$=$	$\ldots$	[RB §3.5, p27]
$Cause$	$=$	$MemAccess \mid DevFault \mid \ldots$	[RB §3.5, p27]
$ha \in HMAction$	$=$	$\ldots$ [RB §3.5, p26]	
$drv_{id} \in DriverId$		—Device Driver Id	[RB §3.7, p28]
$BitField$		—bitfield specification	[RB §3.7, p28]
$RorW$	$=$	$Read \mid Write$	[RB §3.7, p28]
$lport_{id} \in LocPortId$		—Local Port Id	[RB §3.8, p29]

B.2 Key Data Structures

This is basically a hierarchical decomposition of the kernel system into components and sub-components.

$$
\begin{array}{rcll}
PKS \in System & = & Kernel & \\
& \times & Partitions & \text{[RB §3.1, p19]} \\
& \times & XConfig & \text{[RB §2.8, p30]} \\
\\
P \in Partitions & = & PartId \nrightarrow PartRsrc & \text{[RB §3.1, p19]} \\
\\
\mathcal{K} \in Kernel & = & Schedules & \text{[RB §3.1, p20]} \\
& \times & MemMgmt & \text{[RB §3.2, p20]} \\
& \times & IRQMgmt & \text{[RB §3.2, p21]} \\
& \times & TimeMgmt & \text{[RB §3.2, p21]} \\
& \times & IPC & \text{[RB §3.2, p21]} \\
& \times & Health & \text{[RB §3.2, p21]} \\
& \times & Tracing & \text{[RB §3.2, p21]} \\
\\
\mathcal{S} \in Schedules & = & SchedPlanId \nrightarrow Schedule & \text{[RB §4.1, p33]} \\
S \in Schedule & = & Duration \text{ —of MAF (timesteps)} & \text{[RB §3.1, p20]} \\
& \times & TimeSlots & \text{[RB §3.1, p20]} \\
\\
TS \in TimeSlots & = & TimeSlotId \nrightarrow TimeSlot & \text{[RB §3.1, p20]} \\
T \in TimeSlot & = & Start \text{ — (timestep)} & \text{[RB §3.1, p20]} \\
& \times & tdur : Duration \text{ — (timesteps)} & \text{[RB §3.1, p20]} \\
& \times & SchdUnitId & \text{[RB §3.1, p20]} \\
s_{id} \in SchdUnitId & = & PartId \mid (VCPUId \times PartId) \ldots & \text{[RB §3.1, p20]} \\
\\
pres \in PartRsc & = & Memory & \text{[RB §3.3, p21]} \\
& \times & Comms & \text{[RB §3.3, p21]} \\
& \times & VTraps & \text{[RB §3.4.1, p24]} \\
& \times & devs : \mathbb{P} Device & \text{[RB §3.7, p28]} \\
\\
fdir \in Health & = & HMMap & \text{[RB §3.5, p26]} \\
& \times & HMLog^* & \text{[RB §3.5, p26]} \\
hmap \in HMMap & = & PartId \times HMEvent \nrightarrow HMAction & \text{[RB §3.5, p26]} \\
hlog \in HMLog & = & \ldots & \text{[RB §3.5, p26]} \\
\\
dev \in Device & = & DriverId & \text{[RB §3.7, p28]} \\
& \times & (IOPort \times BitField \times RorW) & \\
& \times & irq : IRQ & \\
\\
port_{id} \in PortId & = & PartId \times LocPortId & \text{[RB §3.8, p29]} \\
\\
M \in Memory & = & MemRqd & \text{[RB §3.8, p29]} \\
& \times & \ldots &
\end{array}
$$

$$
\begin{array}{rcll}
C \in Comms & = & \mathbb{P} LocPortId & \text{—sent} \quad \text{[RB §3.8, p29]} \\
& \times & \mathbb{P} LocPortId & \text{—received} \quad \text{[RB §3.8, p29]} \\
& \times & \ldots & \\
IPC & = & PhysPorts & \text{[RB §3.8, p32]} \\
& \times & MsgPaths & \\
 & & & \\
MsgPaths & = & PortId \nrightarrow PortDest & \text{[RB §3.8, p30]} \\
PortDest & = & QueuePortId & \text{—queuing dest. port} \\
& | & Sampl(\mathbb{P} PortId) & \text{—sampling dest.ports} \\
 & & & \\
PhysPorts & = & PhysicalArea & \\
& \times & Port & \\
& \times & Port* & \\
& \times & QueueSize & \\
 & & & \\
Port & = & PortId \times PortAddr \times MsgSize & \\
 & & & \\
XConfig & & \text{—extra config stuff} & \\
& = & ConfigData & \text{[RB §3.8, p31]} \\
& \times & \ldots & \\
 & & & \\
ConfigData & & \text{—coll. of above datatypes} & \\
 & & & \\
CS \in ConfigSpec & & \text{—Configuration File} \quad \text{[RB §4.1, p33]} &
\end{array}
$$

Questions immediately arose:

- $ConfigData$ is a manifestation of a large slice of some of these datatypes. Which one? What kind of slice?

B.3 Key Operations

We also immediately identified some key operations that the kernel would need to be able to perform. Tentative signatures for these functions are presented below. Basically a type signature of the form:

$$op : X \rightarrow System \rightarrow System$$

describes an operator op that takes data of type X as input/command and then operates on the current system state ($System$) to modify it.

$$
\begin{array}{rcll}
initCPU & : & System \rightarrow System & \text{[RB §3.1, p20]} \\
runSched & : & SchedPlanId \rightarrow System \rightarrow System & \text{[RB §3.1, p20]} \\
saveContext & : & PartId \rightarrow System \rightarrow System & \text{[RB §3.1, p20]} \\
restoreContext & : & PartId \rightarrow System \rightarrow System & \text{[RB §3.1, p20]} \\
handleFault & : & HMEvent \rightarrow System \rightarrow System & \text{[RB §3.5, p26]} \\
createPort & : & PortId \rightarrow System \rightarrow System & \text{[RB §3.8, p32]}
\end{array}
$$

B.4 Requirements Summary

These initial developments then stalled. The main reason was that the large scale of this modelling exercise was now becoming very clear, and it was felt that the exploration should focus in on specific parts of interest. Given that ADS was going to explore the use of Frama-C[5] to verify some real kernel code, and concerns raised at meetings regarding the difficulty of doing satisfactory testing of the trap/interrupt handling, it was felt that we should focus on these aspects. These are discussed in Sect. D and Sect. E that follow.

The problem with building a full formal model of the requirements baseline is that it really requires a full-time effort by one of two people to get completed. Doing it on a part-time basis is very difficult simply because, just like modifying programming code not seen for a while, it takes quite a while to get back up to speed.

Is it worth doing? The simplest answer is, if it is required to formally verify the correctness of an implementation w.r.t all of the requirements, then it is absolutely necessary. However, if only certain requirements, or certain aspects of some requirements, are considered as needing formal verification, then it is possible to limit the scope of requirement formalisation accordingly — provided there is confidence in any requirement and/or code impact analysis that has been done.

Another benefit of having a formal (and validated) model of the requirements is that it can be used to analyse them for coherence and consistency.

C Formalising Invariants

In formal software verification, invariants are properties that play a key role in assisting with the task. Simply put, an *invariant* is a property that is always true. In practise this can literally mean it is never false, or can also be interpreted to mean that the invariant needs only be true at certain key points during program execution. In a software context we find two common uses of invariants:

Datatype Invariants are properties that characterise when instances of a datastructure are correctly formed. The simplest practical example of such an invariant is the one typically involved when using binary trees to speed up search: that the binary trees are ordered so that smaller elements lie down the left-subtree, with larger ones to the right. The algorithms that search and build such trees reply on this property in order to work correctly.

Loop Invariants are properties, defined over the values of program variables, that are required to be true when:

1. a program loop is entered for the first time;
2. every time another iteration is about to begin;
3. and immediately upon exiting from the loop

They play a critical role in proving the functional correctness of such loops (a different property, called a *variant*, is used to prove that the loop terminates).

Typically a piece of code is proved correct w.r.t some invariant if it can be shown that if the invariant is true as the code starts, then it remains true when the code completes, even if (especially if!) the code has modified the relevant datatype.

In actual kernel code, there will be a number of implicit datatype invariants associated with various structures, typically implementing such features as lookup tables or queues.

Formal verification will require these to be made explicit, and formalised. Appropriate loop invariants will also need to be discovered and formalised. Tools like Frama-C rely upon a user identifying these invariants (among other properties such as pre- and post-conditions), and annotating the code accordingly.

A key datastructure that dominates the requirements baseline, and indeed the code of any kernel implementation, is that which corresponds to all the kernel *configuration* information. Key to demonstrating kernel correctness is being very clear regarding the corresponding data invariant.

There were two explorations of this invariant conducted as part of this study.

The first was done early on as part of the modelling exercise reported previously (§A), and is discussed next. The second exploration returned back to the requirements much later, and benefited from the knowledge of finer aspects of partitioning kernels and what aspects really matter, that had been acquired as a result of other modelling exercises. This is discussed in Sect. C.2.

C.1 Early Invariant Exploration

We looked at defining the key data invariants that characterised a well-formed configuration. Initially this was done piecemeal, looking at different requirements. Some of the invariant are local to a particular part of the configuration data, whilst others are more global in nature. A few of each were considered at this early exploration statue

NOTE: The function π_i extracts the ith component of a record or tuple, so $\pi_2(a, b, c) = b$, for example. To make this model more readable we really need to replace all uses of π_i with more meaningful names.

C.1.1 The Major Time Frame invariant

The MAF is composed of non-overlapping timeslots [RB §3.1, p20]:

$$
\begin{array}{l}
nonOvlTS : Schedule \to \mathbb{B} \\
nonOvlTS(dur_{MAF}, TS) \\
\quad \widehat{=} \quad totalDur(TS) \leq dur_{MAF} \\
\quad \wedge \quad \forall t_1, t_2 \in \operatorname{dom} TS \bullet t_1 \neq t_2 \implies durRange(t_1) \cap durRange(t_2) = \emptyset
\end{array}
$$

$$
\begin{array}{l}
totalDur : TimesSlots \to Duration \\
totalDur(TS) \\
\quad \widehat{=} \quad \sum_{t \in \operatorname{dom} TS} tdur(TS(t))
\end{array}
$$

$$
\begin{array}{l}
durRange : TimeSlot \to \mathbb{P} Duration \\
durRange(s, d, \_) \\
\quad \widehat{=} \quad \{s, \ldots, s + (d - 1)\}
\end{array}
$$

C.1.2 Interrupt Allocation invariant

A hardware interrupt can only be allocated by configuration to one partition [RB §3.6, p28], and two partitions cannot use the same interrupt line [RB §3.7, p28].

$$
\begin{array}{l}
disjPartIRQ : Partitions \to \mathbb{B} \\
disjPartIRQ(P) \\
\quad \widehat{=} \quad \forall p_1, p_2 \in \operatorname{dom} P \bullet p_1 \neq p_2 \implies (\mathbb{P} irq)(devs(Pp_1)) \cap (\mathbb{P} irq)(devs(Pp_2)) = \emptyset
\end{array}
$$

Configuration data is static, not part of the kernel (?), but cannot be accessed directly by the partitions (so the kernel needs access, and to protect it) [RB §3.8, p31].

C.1.3 Coherent Identifiers invariant

This is a description of a part of a global invariant, that ensures that all identifiers used by one part of the configuration data to refer to another part, are actually defined in that other part.

$$
\begin{array}{l}
coherentIDs : System \to \mathbb{B} \\
coherentIDs(\mathcal{S}, P) \\
\quad \widehat{=} \quad \forall id_S \in \operatorname{dom} \mathcal{S} \bullet \mathbb{P}\pi_3(\operatorname{rng}(\pi_2(\mathcal{S}(id_S)))) \subseteq \operatorname{dom} P
\end{array}
$$

C.1.4 Early Invariants summary

As already indicated at the end of Sect. A, this phase of model building was halted as attention was focussed on interrupts and Frama-C. But this exercise was resumed, pretty much from scratch, once those focussed activities had made a lot of the nature and structure of the kernel much clearer.

C.2 Late Invariant Exploration

Armed with considerably more familiarity with the requirements for the proposed kernel, its target platform, and, to a lesser degree, with the XtratuM code, we revisited the Requirements Baseline and did a systematic trawl through trying to identify key data invariants at a high level of abstraction.

In effect the following list of such invariant properties emerged, presented here in the order in which the need for them was encountered. Not all of them have actually been formalised as of yet. Also, some of the "invariants" found are not static properties of data, but are in fact dynamic or behavioural invariants (i.e., an allowed memory access always succeeds).

C.2.1 Partition Start Address

Partition start address must be in RAM or random-access NVR

C.2.2 Partition Initial Context

Every partition has an initial context, zeroed registers, resetting virtual traps and pending interrupts

C.2.3 Partition Lifecycle

We have the following states and transitions:

$$
\begin{array}{rcl}
INIT(p) & \widehat{=} & start.COLD \rightarrow ACTIVE(p) \\
ACTIVE(p) & \widehat{=} & \neg duringSlot(p)\ \&\ READY(p) \\
 & & \Box duringSlot(p) \& RUNNING(p) \\
 & & \Box hc.Stop.p \rightarrow STOPPED(p) \\
 & & \Box restart.WARM \rightarrow ACTIVE(p) \\
READY(p) & \widehat{=} & slotStart.p \rightarrow RUNNING(p) \\
RUNNING(p) & \widehat{=} & slotEnd.p \rightarrow READY(p) \\
 & & \Box halt.p \rightarrow READY(p) \\
STOPPED(p) & \widehat{=} & start.temp \rightarrow ACTIVE(p)
\end{array}
$$

There are invariants regarding each partition mode:

$$
\begin{array}{l}
\forall p, q \bullet p \neq q \wedge duringSlot(p) \implies \neg duringSlot(q) \\
slotStart.p \rightarrow duringSlot(p)\ \&\ Q = slotStart.p \rightarrow Q \\
slotEnd.p \rightarrow \neg duringSlot(p)\ \&\ Q = slotEnd.p \rightarrow Q
\end{array}
$$

C.2.4 Authorised Starter Partition

There exists at least one partition authorised to start other partitions

C.2.5 Partition Memory Separation

No two partitions should share the same memory addresses (Exception: shared messaging space? e.g. "zero copy"?)

C.2.6 Allowed Access always succeeds

A memory access by a partition to its authorised memory always succeeds

C.2.7 Forbidden Access always fails

A memory access by a partition to memory for which it has no authorisation always fails

C.2.8 Memory Access Permissions are fixed

The memory access permissions for a partition are time-invariant

C.2.9 Partition Time Separation

Basically every partition has a sequence of timeslots, usually defined as by start/stop time offsets within a major time frame. We can model these as sets of times (or "timesteps")

$$
\begin{array}{rcl}
\mathcal{S} & : & Partition \rightarrow \mathcal{P}\, Time \\
p \neq q & \implies & \mathcal{S}(p) \cap \mathcal{S}(q) = \emptyset
\end{array}
$$

C.2.10 Minimum Timestep

$$
timestep \geq TIMESTEP\_DURATION
$$

C.2.11 Timeslot smaller than MAF

$$\max \mathcal{S}(p) \quad \leq \quad \text{MAF\_DURATION}$$

C.2.12 Scheduler Invariant (?)

This forces an elaboration in the scheduler:

$$\mathcal{S} \quad : \quad Partition \to \mathcal{P}(Slot \times \mathcal{P}\, Time)$$

$$\begin{array}{rcl} \mathcal{S}(p) = \{(s, \_), (t, \_), \ldots\} & \implies & s \neq t \\ (s, \_) \in \mathcal{S}(p) \land (t, \_) \in \mathcal{S}(q) & \implies & s \neq t \end{array}$$

We realise that there is a functional constraint that suggests are reworking follows:

$$\mathcal{S} \quad : \quad Slot \to (Partition \times \mathcal{P}\, Time)$$

$$\mathcal{S}(s) = (p, T) \land \mathcal{S}(t) = (q, U) \quad \implies \quad p \neq q \land T \cap U = \emptyset$$

C.2.13 Schedule Plans invariant

$$SS \quad : \quad Plan \to (Slot \to Partition \times \mathcal{P}\, Time)$$

Are slot-ids unique across plans as well, or is the pair $Plan \times Slot$ the unique identifier?

C.2.14 No Virtual Interrupts if not running

Partitions that have yielded have their virtual interrupts masked

C.2.15 Port ownership invariant

Each port is owned by one partition: $Port \to Partition$

C.2.16 Allowed Access always succeeds

Each port has four attributes:

$$Port \quad = \quad Name \times Dir \times Mode \times Size$$

This turns out to be strictly false — there is another numeric attribute depending on the mode

C.2.17 Unique Port Names

Port names as unique within a partition.

$$PP \quad : \quad Partition \to Name \to Dir \times Mode \times Size$$

However, portnames need to be global!

C.2.18 Port complementarity

For every source (destination) port in one partition there must exist at least one corresponding destination (source) port in another (or same!) partition (See notes in **PK-34**).

$$PP(P)(n) = (d, m, s) \quad \implies \quad \exists Q \bullet PP(Q)(n) = (opp(d), m, s)$$

Here *opp* maps "source" to "destination" and *v.v.*

C.2.19 One source per channel

For any port there is exactly one partition in which it is a source port

$$n \in \operatorname{dom} PP(P) \quad \Longrightarrow \quad \exists! Q \bullet n \in \operatorname{dom} PP(Q) \land \pi_1(PP(Q)(n)) = \text{source}$$

From **RB §3.8, p29** we see that "each message should be unique and identifiable", with the "only means" being the "port identifier, which is local to the partition".

C.2.20 IPC Invariant?

We have logical LINKS, and NAMEed PORTs, with PORTs owned by one partition. Can we conflate some of these concepts?
We have a blurred distinction between IPC "links", "ports" owned by partitions, and "names" unique within a partition. Are "names" the way we associate "ports" to form "links"? For now we assume the answer to the above is yes.

$$\begin{array}{rcl} TMode & = & \textsc{Sample}\ Time\ Partition\ Partition \\ & | & \textsc{Queue}\ Size\ Partition\ (\mathcal{P}\, Partition) \\ IPC & = & Name \rightarrow Size \times TMode \end{array}$$

C.2.21 Mode constraints for Destination ports

$$\begin{array}{rcl} \textsc{Sample}\ t\ p\ P & = & p \notin P \land P \neq \emptyset \\ \textsc{Queue}\ s\ p_1\ p_2 & = & s > 0 \land p_1 \neq p_2 \end{array}$$

$$Partition \rightarrow \mathcal{P}\, Interrupt$$

C.2.22 Physical Interrupt Separartion

Different physical interrupts can be routed to one partition, but one physical interrupt cannot be routed to different partitions.

C.2.23 Health Monitoring sees all

Kernel configuration should ensure that all the above traps and errors are passed to HM.

$$HMLog \quad = \quad Date \times HMLogType \times (Partition | \textsc{Kernel}).$$

C.2.24 Partition interrupts masked when not running

$$\neg running(p) \quad \Longrightarrow \quad \forall int \bullet isAlloc(int, p) \implies isMasked(int).$$

C.2.25 Taking Stock

In essence we have a long list of various properties that must hold for a valid configuration. What needs to be done is ensure that all the ad-hoc formal notation used here is coherent, well-defined, and all talks about the same conceptual thing. Then the final grand invariant is simply a giant conjunction (logical-and) of all the small invariant pieces.

However, we also have some unanswered questions regarding how ports in different partitions are linked to form channels. It was not clear precisely how ports from different partition were linked together. We decided to look at some of the example Xtratum XML config files to see how they did it.

C.3 Ports and Channels in Xtratum XML Config Files

The situation in the Xtratum config files is a little different. The Xtratum distribution comes with some examples, each of which lives in its own folder, which contains a file called `xm_cf.sparcv8.xml` that is the configuration.

In folder `user/xal/examples/example.003` we find such a file from which we show two extracts. The first is how the partitions are declared:

```
<PartitionTable>
  <Partition id="0" flags="system" name="Partition1" console="Uart">
    <PhysicalMemoryAreas>
      <Area start="0x40180000" size="128KB"/>
    </PhysicalMemoryAreas>
    <TemporalRequirements duration="200ms" period="500ms"/>
    <PortTable>
      <Port name="sport1_w" type="sampling" direction="source" />
      <Port name="sport2_r" type="sampling" direction="destination" />
      <Port name="sport3_r" type="sampling" direction="destination" />
    </PortTable>
  </Partition>

  <Partition id="1" name="Partition2"   console="Uart">
    <PhysicalMemoryAreas>
      <Area start="0x401A0000" size="128KB"/>
    </PhysicalMemoryAreas>
    <TemporalRequirements duration="200ms" period="500ms"/>
    <PortTable>
      <Port name="sport1_r" type="sampling" direction="destination" />
      <Port name="sport2_w" type="sampling" direction="source" />
    </PortTable>
  </Partition>

  <Partition id="2" name="Partition3"    console="Uart">
    <PhysicalMemoryAreas>
      <Area start="0x401c0000" size="128KB"/>
    </PhysicalMemoryAreas>
    <TemporalRequirements duration="100ms" period="500ms"/>
    <PortTable>
      <Port name="sport1_r" type="sampling" direction="destination" />
      <Port name="sport3_w" type="sampling" direction="source" />
    </PortTable>
  </Partition>
</PartitionTable>
```

Each partition declares its ports using locally unique names:

$$
\begin{aligned}
p \in XPNo &= \mathbb{N} \\
m \in XMode &= XSAMPLE \mid XQUEUE \\
d \in XDir &= XSRC | XDST \\
ps \in XPort &= Name \to XMode \times XDir \\
\rho \in XPartition &= XPNo \to XPort \times \ldots
\end{aligned}
$$

The second extract shows "channels" being declared

```
<Channels>
  <SamplingChannel maxMessageLength="12B" refreshPeriod="1s">
    <Source partitionId="0" portName="sport1_w"/>
    <Destination partitionId="1" portName="sport1_r"/>
    <Destination partitionId="2" portName="sport1_r"/>
  </SamplingChannel>
  <SamplingChannel maxMessageLength="12B" refreshPeriod="1s">
    <Source partitionId="1" portName="sport2_w"/>
    <Destination partitionId="0" portName="sport2_r"/>
  </SamplingChannel>
  <SamplingChannel maxMessageLength="12B" refreshPeriod="1s">
    <Source partitionId="2" portName="sport3_w"/>
    <Destination partitionId="0" portName="sport3_r"/>
  </SamplingChannel>
</Channels>
```

So a collection of unnamed channels indicate how they all connect up

$$
\begin{array}{rcl}
md \in XModeDims & = & XCSAMPLE\ Time\ Size \mid XCQUEUE\ Length\ Size \\
cp, (d, p, n) \in XCHPort & = & XDir \times XPNo \times Name \\
ch, (md, cps) \in XChannel & = & XModeDims \times XCHPort+ \\
\kappa \in XChans & = & \mathcal{P}\, XChannel
\end{array}
$$

We assume that $Name$, $Size$ and $Length$ are obvious types.

We have a complex invariant here that needs to ensure that both structures are mutually and individually consistent:

- For every partition p in ρ, each port name is unique. This is guaranteed structurally by our modeling the relationship as a function from port names to port information.
- For every channel ch in κ, there is exactly one source port, and the number of destination ports is consistent with the mode.

$$
\begin{array}{rcl}
\forall (md, cps) \in \kappa & \bullet & validCh(md, cps) \\
validCh(XCSAMPLE\ \_\ \_, cps) & \widehat{=} & \#cps \geq 2 \wedge hasOneSrc(cps) \\
validCh(XCQUEUE\ \_\ \_, cps) & \widehat{=} & \#cps = 2 \wedge hasOneSrc(cps) \\
hasOneSrc(cps) & \widehat{=} & \#(filter_{isSrc} cps) = 1 \\
isSrc(d, p, n) & \widehat{=} & d = XSRC
\end{array}
$$

- Every port in ρ has a corresponding consistent entry in κ.

$$
\begin{array}{rcl}
\forall p \in \rho & \bullet & \textbf{let}\ (ps, \dots) = \rho(p)\ \textbf{in} \\
\forall n \in ps \bullet \exists (md, cps) \in \kappa, ch \in cps & \bullet & pMatch(ps(n), md, ch)
\end{array}
$$

- For every channel definition in κ, each port has a corresponding consistent entry in ρ.

$$
\begin{array}{rcl}
\forall (md, cps) \in \kappa \bullet \exists p \in \rho & \bullet & \textbf{let}\ (ps, \dots) = \rho(p)\ \textbf{in} \\
\exists n \in ps, ch \in cps & \bullet & pMatch(ps(n), md, ch)
\end{array}
$$

The last two checks depend on the following:

$$
\begin{array}{rcl}
pMatch((m, d_p), md, (d_c, \dots)) & \widehat{=} & d_p = d_c \wedge mMatch(m, md) \\
mMatch(XSAMPLE, XCSAMPLE \dots) & \widehat{=} & \textbf{true} \\
mMatch(XQUEUE, XCQUEUE \dots) & \widehat{=} & \textbf{true} \\
mMatch(\_, \_) & \widehat{=} & \textbf{false}
\end{array}
$$

So, by looking at the way IPC is described in the XtratuM config file, we have exposed both a vaqueness in the baseline requirements regarding ports, links and channels, as well as what appears to be some missing requirements in this area. It was the attempt in the previous section to formalise that exposed the missing information.

C.4 Ports and Channels in XtratuM code

We can dive even deeper and look for the corresponding declarations of C datatypes in the XtratuM code.

In `user/tools/xmcparser/xmc.h` we find:

```
1  struct xmcCommChannelNoL {
2      int type;
3      union {
4     struct {
5         int maxLength;
6         int maxNoMsgs;
7     } q;
8     struct {
9         int maxLength;
10    } s;
11     };
12     int refreshPeriod;
13 };
14
15 struct xmcCommPortNoL {
16     int name;
17     int direction;
18     int type;
19 };
20
21 struct ipcPort {
22     int channel;
23     char *partitionName;
24     char *portName;
25     xm_u32_t partitionId;
26     int direction;
27 };
28
29 struct ipcPortNoL {
30     int partitionName;
31     int portName;
32     int partitionId;
33     int direction;
34 };
```

These seem to correspond to parts of the XML in the previous section. Interestingly, it seems that a `refreshPeriod` is associated with both types of ports: sampling and queueing. This contradicts the Requirements baseline which only associates refresh rates with sampling ports. It is also very unclear what the `NoL` suffix is intended to indicate.

In `core/include/xmconf.h` we learn little more about communication ports:

```
struct xmcCommPort {
    xm_u32_t nameOffset;
    xm_s32_t channelId;
#define XM_NULL_CHANNEL -1
    xm_s32_t direction;
#define XM_SOURCE_PORT 0x2
#define XM_DESTINATION_PORT 0x1
    xm_s32_t type;
#define XM_SAMPLING_PORT 0
#define XM_QUEUING_PORT 1
    xm_u32_t flags;
#define XM_FIFO_PORT 0
#define XM_PRIORITY_PORT 1
};
```

and about partitions:

```
struct xmcPartition {
    xmId_t id;
    xm_u32_t nameOffset;
    xm_u32_t flags;
#define XM_PART_SYSTEM 0x100
#define XM_PART_FP 0x200
    xm_u32_t hwIrqs;
    xm_s32_t noPhysicalMemoryAreas;
    xm_u32_t physicalMemoryAreasOffset;
    xmDev_t consoleDev;
    struct xmcTemporalRestrictions {
   xm_u32_t period;
   xm_u32_t duration;
    } temporalRestrictions;
    //struct xmcPartitionArch arch;
    xm_u32_t commPortsOffset;
    xm_s32_t noPorts;
    struct xmcHmSlot hmTab[XM_HM_MAX_EVENTS];
    xm_u32_t ioPortsOffset;
    xm_s32_t noIoPorts;
    struct xmcTrace trace;
};
```

and about channels:

```
struct xmcCommChannel {
#define XM_SAMPLING_CHANNEL 0
#define XM_QUEUING_CHANNEL 1
    xm_s32_t type;
   xm_s32_t maxLength;
   xm_u32_t maxNoMsgs_refreshPeriod;
   xm_s32_t maxTimeExpiration_noReceivers;
};
```

But, which of these is the runtime representation of a concept? Is there a difference between datatypes to hold configuration data, and those that handle the implementation of the actual feature? It does appear that data-structures with prefix `xmc` are part of the runtime

configuration data, but it would be useful to have this confirmed.

Finally, in `core/include/objects/commports.h` we find

```
typedef struct {
    char *__gParam portName;
#define XM_INFINITE_TIME ((xm_u32_t)-1)
    xmTime_t refreshPeriod; // Refresh period.
    xm_u32_t maxMsgSize; // Max message size.
    xm_u32_t direction;
} xmSamplingPortInfo_t;

typedef struct {
    char *__gParam portName;
    xm_u32_t maxMsgSize; // Max message size.
    xm_u32_t maxNoMsgs; // Max number of messages.
    xm_u32_t direction;
    xm_u32_t flags;
} xmQueuingPortInfo_t;

union channel {
    struct {
   char *buffer;
   xm_s32_t length;
   xmTime_t timestamp;
        kThread_t **receiverTab;
        xm_s32_t *receiverPortTab;
        xm_s32_t noReceivers;
        kThread_t *sender;
        xm_s32_t senderPort;
    } s;
    struct {
   struct msg {
       struct dynListNode listNode;
       char *buffer;
       xm_s32_t length;
       xmTime_t timestamp;
   } *msgPool;
   struct dynList freeMsgs, recvMsgs;
   xm_s32_t usedMsgs;
   kThread_t *receiver;
        xm_s32_t receiverPort;
        kThread_t *sender;
        xm_s32_t senderPort;
    } q;
};

struct port {
    xm_u32_t flags;
#define COMM_PORT_OPENED 0x1
#define COMM_PORT_EMPTY 0x0
#define COMM_PORT_NEW_MSG 0x2
#define COMM_PORT_CONSUMED_MSG 0x4
#define COMM_PORT_MSG_MASK 0x6
    xmId_t partitionId;
};
```

This all looks like part of the IPC implementation.

What is clear here is that a high-level requirements invariant, for example that a queueing channel can only have one destination port, has to be reflected in low-level datatype invariants in diverse parts of the code. In fact a major problem with the XtratuM sources *as given*, is the complete lack of any (obvious) high-level overview or notion of kernel design. The other problem is that the sources are designed to be a single repository for all code that ports XtratuM to a wide range of platforms. It is very hard to establish precisely which bits of the code will end up being part of an actual kernel running on the IMAKQP target [platform, namely the LEON2+MPU/LEON3 Sparcv8 architectures.

D Formalising Interrupts

Traps and interrupts play a crucial role in the operation of a separation kernel. The kernel configures the target platform to ensure that appropriate traps are raised when events of interest to the kernel occur. These include alerts from the MMU regarding invalid memory accesses, or expected I/O device interrupts. What is allowable varies depending on which partition is currently running and whati autorisations it has from the configuration. A partition's only way to invoke kernel services such as IPC is by using a trap instruction. Kernel correctness depends on it ensuring that the right settings in hardware are in place at the right time. Add to this the fact that the timing of interrupts is effectively non-deterministic, originating often from hardware running in parallel with the CPU, then we can see that there is quite a verification challenge, regardless of which method is being used.

Here we discuss ongoing work looking at how me might formally prove that the kernels setup and handling of traps is correct, in that it satisfies the requirements.

D.1 Basic Formalisation

Some aspects of the trap infrastructure is quite easy to formalise, for example the terminology used in D02:

TRAP vectored transfer of control through trap table

SWTRAP Software Trap –trap deliberately induced by an instruction

EXCPT Exception — trap caused by non-trap instruction as a result of a runtime problem (dividend zero, page fault, register window overflow)

- **NOMINAL** Exception encountered as part of normal operation, e.g. page fault or register window overflow
- **ERROR** Exception as a result of erroneous instruction usage, e.g., divide by zero, or bad memory access.

HWINT Hardware Interrupt — trap generated by a request from external hardware: IO devices, timers.

$$\begin{aligned} TRAP &= SWTRAP \mid EXCPT \mid HWINT \\ EXCPT &= NOMINAL \mid ERROR \end{aligned}$$

A trap is virtualized if it leads to a partition-provided handler being invoked. The *variety* of traps, both physical and virtual requires careful modelling.

We use [Leon §M.N, pX] to refer to Section M.N, page X of *LEON3-FT SPARC V8 Processor LEON3FT-RAX Data Sheet and User's Manual*. From that we can formalise interrupt levels and masking

The LEON3FT has a 15-bit IRQ pending bus, with a system level priority register (0 or 1) and processor level force and mask registers. and each processor can both force and mask each interrupt line. The highest priority interrupt is passed through [Leon §9.2.1, p77]:

$$
\begin{array}{rcll}
Irq, Prio \in IRQ & = & \mathbb{P}\{1,\ldots,15\} & \\
cpu \in CPUId & & \text{—CPU Identifier} & \\
force \in FORCE & = & CPUId \to IRQ & \\
mask \in MASK & = & CPUId \to \{1,\ldots,14\} & \text{[Leon §9.2.1, p78]} \\
raise & : & CPUId \to IRQ \to \{0,\ldots,15\} & \\
raise(cpu)(Irq) & \widehat{=} & pencode((Irq \cup force(cpu)) \cap (mask(cpu) \cup \{15\}) & \\
pencode & : & IRQ \to \{0,\ldots,15\} & \\
pencode(Irq) & \widehat{=} & \textbf{let } Hi = Irq \cap Prio & \\
 & & \textbf{in if } Hi \neq \emptyset & \\
 & & \textbf{then } encode(Hi) & \\
 & & \textbf{else } encode(Irq) & \\
encode & : & IRQ \to \{0,\ldots,15\} & \\
encode(\emptyset) & \widehat{=} & 0 & \\
encode(Irqs) & \widehat{=} & \max Irqs &
\end{array}
$$

$$
\begin{array}{rcll}
ack & : & IRQ \to (IRQ \times FORCE) \to (IRQ \times FORCE) & \\
ack(irq)(Irqs, Force) & & & \\
 & \widehat{=} & \textbf{if } irq \in Force & \text{[Leon §9.2.1, p78]} \\
 & & \textbf{then } (Irqs, Force \setminus \{irq\}) & \\
 & & \textbf{else } (Irqs \setminus \{irq\}, Force) &
\end{array}
$$

$$
\begin{array}{rcll}
reset & : & (IRQ \times FORCE \times MASK) & \\
reset & \widehat{=} & (\_, \emptyset, \_) & \text{[Leon §9.2.1, p78]}
\end{array}
$$

Underneath all the complexity[1] we have a fairly standard trap/interrupt model.
In [RB §3.6, pp27] we see Figure 3.6-1, which shows how interrupts are managed. Notable by their absence are nominal exceptions.

$$
\begin{array}{rcll}
Vector_n\ A & = & \{0,\ldots,n-1\} \to A & \\
vtt \in VTraps & = & Vector_{15?}TH & \text{[RB §3.6, p27]} \\
th \in TH & & \text{—Trap Handler} &
\end{array}
$$

These are all quite simple models of what is either structural information or straightforward functional behaviour. Most of this can be treated simply and abstractly.

D.2 Concurrency and Timing

The real challenge arises because we have a number of hardware entities (CPU,MMU,MEM,DEV,IRQ) all running in parallel (Fig. 5). Three (MMU,MEM and IRQ) are passive in that they respond to external events, and based on their current configuration, they determine how to respond. Two (CPU and DEV) are active in that they "decide" what is going to happen. The CPU's decisions come from the program it is executing, while the DEV is passing on I/O events that arise from outside the system. The key feature here is both CPU and DEV are effectively independent, the the DEV behaviour looks completely non-deterministic from the perspective of the CPU.

[1] sliding register windows

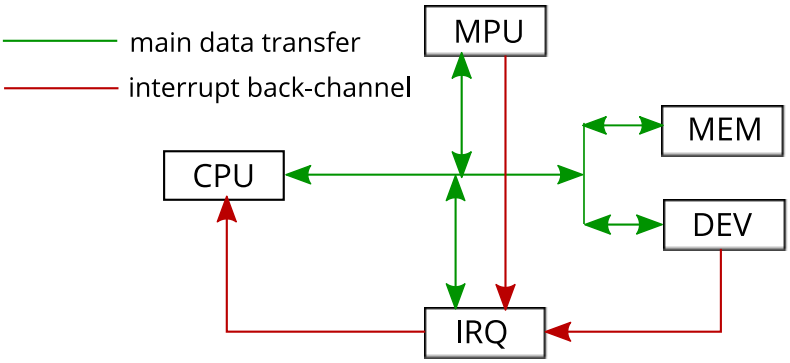

Figure 5: PK Physical Hardware and Signal links

The CPU's responsibility (when running kernel code) is to ensure that MMU, DEV and IRQ are configured to that when it reverts back to partition code, then IRQ properly handles both anything flagged, correctly, by the MMU, and anything, no matter how unexpected that comes from DEV. When IRQ raises trap with the CPU, then the handlers, which are considered as kernel code, must do the correct thing.

D.3 Modelling Approach

At present we are using the modelling language called Communicating Sequential Processes (CSP)[4] to model (i) the physical hardware as shown in Fig.5; and (ii) an abstract view of the running kernel and partition software; and (iii) abstract models of the baseline requirements. The plan is to validate the model by showing that the interaction of these two models satisfies the baseline requirements. In CSP this is typically achieved by putting the various models in parallel, specifying how they must synchronise, and then verifying that the combined model has key properties. A common approach is either to show that one model refines another (this can be used to verify both liveness and safety properties), while another is to setup the combined model so that if the property is true then the model is deadlock-free, while falsity is indicated by a deadlock with the path that leads to the deadlock being a counter-example.

We do not intend to prove any such properties, but will rather make use of the FDR3 refinement-checking tool[15, 16] for doing automated exhaustive testing of the models.

This work is being done in collaboration with a Masters student at TCD, Kevin Hennessy, and forms part of his dissertation.

D.3.1 Hardware Model

The approach adopted is to start with the hardware model, and try to find the simplest model that has all the relevant features. Relevance here means that our main concern is being able to capture all the pertinent interrupt handling behaviour. The output of this exercise will be a model of behaviour characterised by visible events that model data and signals that pass between the hardware components.

The main issue being addressed in this model is accurately capturing *when* the CPU responds to a trap raised by the IRQ unit. The real processor has a 7-stage pipeline, which checks for interrupts at the 6th stage. We have compressed the first 5 stages into one, as nothing externally visible occurs then.

Another key challenge is ensuring that all the internal components are kept in sync, with an appropriate use of clock events.

The following extract shows the current state of the pipeline model, which is currently over-elaborate (which helped while debugging the model). We are now slimming the model down to the simplest form that retains essential features.

```
IU_PIPELINE(a, b, c) = ptick -> ctxPeek?pid -> (
   fe.pid?d -> p_exception!b -> p_exception2
    -> exec!c -> exec2 -> wr!a -> wr2 -> IU_PIPELINE(b, c, d)
   []
   ctxPeek?debugInfo -> returnFromHandler -> ctxPop
    -> ctxPeek?debugInfo -> p_exception!b -> p_exception2
    -> exec!c -> exec2 -> wr!a -> wr2 -> IU_PIPELINE(b, c, nop)
)

IU_WRITE = wr?(instID)
 -> if instID == writeInst
    then tr.w.a0 -> ( memwrite.d0 -> tr2 -> wr2 -> IU_WRITE
                    []
                    tr_abort -> tr2 -> wr2 -> IU_WRITE )
    else ( wr2 -> IU_WRITE )

IU_CONTEXT(context,cmax)
 =  ctxPeek!(head(context)) -> IU_CONTEXT(context,cmax)
    []
    ctxPush?new_pid -> if length(context) >= cmax
                       then ( contextFull -> IU_CONTEXT(context,cmax) )
                       else ( IU_CONTEXT(<new_pid>^context,cmax)
    []
    ctxPop -> if length(context) ==1
              then ( IU_CONTEXT(context,cmax) )
              else ( IU_CONTEXT(tail(context),cmax) )


....

-- assembling pipeline
INTERNAL_CPU0 = IU_PIPELINE(nop, nop, nop)
INTERNAL_CPU1 = INTERNAL_CPU0
                [| {|ctxPeek , returnFromHandler |} |]
                IU_CONTEXT(ctx_initial,cmax)

INTERNAL_CPU2 = INTERNAL_CPU1 [| {| wr, wr2 |} |] IU_WRITE


....
```

Here `a -> P` is a process that does event/action `a` and then behaves like process `P`. Notation `a?x -> P(x)` is an input action where a value is input on channel `a` and stored in variable `x`, before running `P`. To output value of expression `e` on channel `a` use `a!e -> P` Process `(a -> P [] b -> Q)` is a process offers a choice between `P` and `Q` to the environment which selects the one to run by performing either action `a` or action `b`. We put `P` and `Q` in parallel by specifying on which events/actions they must synchronise by writing `P [| synch-events |] Q`. The notation `{| chans |}` describes all events associated with the listed channels

D.3.2 Kernel & Partitions Model

This will be followed up with the abstract kernel/partition model that will be tailored to fit the hardware model. This should be very straightforward because we have abstracted out programs in the hardware model. Rather than having the CPU get instructions by reading from MEM, we simply supply these from a separate process that effectively streams a sequence of instructions to the CPU. In effect the abstract model will be a number of such instruction streaming processes, one of which will be designated as the kernel.

D.3.3 Requirements Model

This will be a collection of small processes that model expected behaviour patterns. Typically they will describe a partition trying to do something and then either being let continue, if so authorised, or showing the kernel intervening and the appropriate HM response.

D.4 Summary

This is still very much a work in progress. The model is being worked on by Kevin Hennessy, a fifth-year student doing a Masters Dissertation, the due date for which is May 10th 2016. Some of this will be reported on at DASIA 2016 on May 12th. We would expect a final version to be in the report submitted for the Final Review in early June 2016.

We include here a listing to show how it is done:

```
1
2
3
4  {- Clock -}
5
6  channel tick --used for syncing each hardware component
7  CLOCK = tick -> CLOCK
8
9
10 {- Memory and Bus -}
11
12 --Memory is represented by a map where Mem_Region is a key for Data
13
14 datatype CodeOwner = jDC | kernel | partition1 | partition2 | mf_handler | supI_handler
   | scheduler
15 --(DC stands for DontCare)
16
17
18 --convention - aX is always forbidden to everyone, even the kernel (for debugging)
19 datatype Mem_Region = a0 | aDC | aX
20
21 datatype Data = d0  | dDC
22
23 datatype Op = nopInst | writeInst | mmuBlockInst | mmuUnBlockInst | setSupModeInst |
   setUsrModeInst | jumpInst | handlerReturnInst
24
25
26
27
28 mem_init = (| a0 => d0, aDC => d0, aX => d0 |)
29
```

```
--supervisor mode only instructions and memory regions
supervisor_instructions = {mmuBlockInst,mmuUnBlockInst, setSupModeInst}


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)) )



{-Memory Protection Unit -}

channel raise, raiseFailed : InTag  --synced with IRQ

channel mmuBlock, mmuUnBlock : Mem_Region    --synced with CPU

mmu_init= {aX}

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}))
```

```
{- Interrupts and Interrupt Controller -}



datatype InTag = memfault | supInst | schedulerInterrupt

interruptVector = <
                  (supInst, false),
                  (memfault, false),
                  (schedulerInterrupt, false)
                                    >  --(list in order of priority high to low)


--raiseFailed  -- synced with MMU / Device
--raise        -- synced with MMU / Device

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) )


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))
          )
   )

{-Interrupt Helper Functions -}

--raise an interrupt
raise_interrupt(tag, vector) = set_bit(tag, <>, 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. hen you call this function you assume there
  else if is_irq_activated(head(sublist2)) == true
       then get_irq_tag(head(sublist2))
       else get_next_tag(<head(sublist1)>, tail(sublist2) )

get_irq_tag( (tag, status) )
= tag




{- CPU -}
channel irqFwdDisabled --not synced with anything, just for info

--Return popped stack. Keep the bottom element (kernel) on the bottom of the stack always
popped_stack(stack) = if length(stack) ==1
                      then ( stack )
                      else ( tail(stack) )


channel fe : CodeOwner.Op.CodeOwner.Mem_Region.Data --synced with CODE.



channel BADINTERRUPTTAG --show status before STOP

```

```
201
202 CPU(currLoc, currSupBit, lastLoc, lastSupBit, TrapsEnabled)--(currLoc, currSupBit, lastLoc, lastSup
203
204 = tick -> SYNC_DEVICES(currLoc, currSupBit, lastLoc, lastSupBit, TrapsEnabled)
205
206 --used to make sending interrupts from DEVICES to IRQ more deterministic
207 SYNC_DEVICES(currLoc, currSupBit, lastLoc, lastSupBit, TrapsEnabled)
208 = devStart -> devEnd -> CHECK_IRQ_ENABLED(currLoc, currSupBit, lastLoc, lastSupBit, TrapsEnabled)
209
210 CHECK_IRQ_ENABLED(currLoc, currSupBit, lastLoc, lastSupBit, TrapsEnabled)
211 = ( if TrapsEnabled == false
212     then  ( irqFwdDisabled -> FETCH(currLoc, currSupBit, lastLoc, lastSupBit, TrapsEnabled)
)
213     else ( CHECK_INTERRUPTS(currLoc, currSupBit, lastLoc, lastSupBit, TrapsEnabled) )
214   )
215
216 --channel test
217 CHECK_INTERRUPTS(currLoc, currSupBit, lastLoc, lastSupBit, TrapsEnabled)
218 = (noPendingIRQs -> FETCH(currLoc, currSupBit, lastLoc, lastSupBit, TrapsEnabled) )
219
220   [] ( irqForward?tag ->
221      ( --this block of code is equivilent to trap table (matching interrupts to handler code)
222      if tag == memfault
223      then ( HANDLE(currLoc, currSupBit, lastLoc, lastSupBit, TrapsEnabled, mf_handler) )
224      else ( if tag == supInst
225             then ( HANDLE(currLoc, currSupBit, lastLoc, lastSupBit, TrapsEnabled, supI_handler) )
226             else ( if tag == schedulerInterrupt
227                    then ( HANDLE(currLoc, currSupBit, lastLoc, lastSupBit, TrapsEnabled,
scheduler) )
228                    else( BADINTERRUPTTAG -> STOP )
229                  )
230           )
231      )
232      )
233
234 HANDLE(currLoc, currSupBit, lastLoc, lastSupBit, TrapsEnabled,handler)
235 =  FETCH(handler, true, currLoc, currSupBit, false)
236
237
238 FETCH(currLoc, currSupBit, lastLoc, lastSupBit, TrapsEnabled)
239 = (fe.(currLoc)?inst -> CHECK_SUP_BIT_INST(currLoc, currSupBit, lastLoc, lastSupBit, TrapsEnabled,
240
241
242 get_owner((owner, mode)) = owner
243
244 get_mode((owner, mode)) = mode
245
246
247 CHECK_SUP_BIT_INST(currLoc, currSupBit, lastLoc, lastSupBit, TrapsEnabled, op.jmp.addr.dat )
248 =  (if currSupBit == false
249    then (if member(op, supervisor_instructions)
250          then ( raise!supInst -> CPU(currLoc, currSupBit, lastLoc, lastSupBit, TrapsEnabled) )
251          else EXEC_RETURN_INST(currLoc, currSupBit, lastLoc, lastSupBit, TrapsEnabled, op.jmp.addr.d
252
253   else EXEC_RETURN_INST(currLoc, currSupBit, lastLoc, lastSupBit, TrapsEnabled, op.jmp.addr.dat ) )
254
255
```

```
256
257 EXEC_RETURN_INST(currLoc, currSupBit, lastLoc, lastSupBit, TrapsEnabled, op.jmp.addr.dat )
258 = if op == handlerReturnInst
259   then ( --enable IRQs and pop stack
260          CPU(lastLoc, lastSupBit, currLoc, currSupBit, true)
261        )
262   else ( EXEC_JUMP_INST(currLoc, currSupBit, lastLoc, lastSupBit, TrapsEnabled, op.jmp.addr.dat ) )
263
264
265
266
267 EXEC_JUMP_INST(currLoc, currSupBit, lastLoc, lastSupBit, TrapsEnabled, op.jmp.addr.dat )
268 = if op == jumpInst
269   then ( --CPU(jmp, currSupBit, currLoc, currSupBit, true)
270         if jmp == jDC
271         then ( CPU(currLoc, currSupBit, lastLoc, lastSupBit, TrapsEnabled) ) --jump to DONTCARE is a
272         else ( CPU(jmp, currSupBit, currLoc, currSupBit, true) )
273        )
274
275   else ( EXEC_SUPMODE_INST(currLoc, currSupBit, lastLoc, lastSupBit, TrapsEnabled, op.jmp.addr.dat
)
276
277
278  EXEC_SUPMODE_INST(currLoc, currSupBit, lastLoc, lastSupBit, TrapsEnabled, op.jmp.addr.dat )
279  = (if op == setSupModeInst
280     then CPU(currLoc, true, lastLoc, lastSupBit, TrapsEnabled)
281     else EXEC_USRMODE_INST(currLoc, currSupBit, lastLoc, lastSupBit, TrapsEnabled, op.jmp.addr.dat
282
283
284 EXEC_USRMODE_INST(currLoc, currSupBit, lastLoc, lastSupBit, TrapsEnabled, op.jmp.addr.dat )
285  = (if op == setUsrModeInst
286     then CPU(currLoc, false, lastLoc, lastSupBit, TrapsEnabled)
287     else EXEC_MMU_BLOCK_INST(currLoc, currSupBit, lastLoc, lastSupBit, TrapsEnabled, op.jmp.addr.da
288
289
290 EXEC_MMU_BLOCK_INST(currLoc, currSupBit, lastLoc, lastSupBit, TrapsEnabled, op.jmp.addr.dat )
291 = (if op == mmuBlockInst
292    then mmuBlock!addr -> CPU(currLoc, currSupBit, lastLoc, lastSupBit, TrapsEnabled)
293    else EXEC_MMU_UNBLOCK_INST(currLoc, currSupBit, lastLoc, lastSupBit, TrapsEnabled, op.jmp.addr.d
)
294
295 EXEC_MMU_UNBLOCK_INST(currLoc, currSupBit, lastLoc, lastSupBit, TrapsEnabled, op.jmp.addr.dat )
296 = (if op == mmuUnBlockInst
297    then mmuUnBlock!addr -> CPU(currLoc, currSupBit, lastLoc, lastSupBit, TrapsEnabled)
298    else EXEC_WRITEINST(currLoc, currSupBit, lastLoc, lastSupBit, TrapsEnabled, op.jmp.addr.dat )
)
299
300
301 EXEC_WRITEINST(currLoc, currSupBit, lastLoc, lastSupBit, TrapsEnabled, op.jmp.addr.dat )
302 = (if op == writeInst
303    then ( if addr == aDC
304           then ( CPU(currLoc, currSupBit, lastLoc, lastSupBit, TrapsEnabled)) --nop
305           else ( bus.w.addr ->((write!dat -> CPU(currLoc, currSupBit, lastLoc, lastSupBit, TrapsEnab
306                                []((CPU(currLoc, currSupBit, lastLoc, lastSupBit, TrapsEnabled)))
) )
307    else CPU(currLoc, currSupBit, lastLoc, lastSupBit, TrapsEnabled) )
```

E Using Frama-C

All our exploration of actual kernel code was based on the XtratuM code version v3.3-1.3 that we had obtained for the prior MTOBSE project. Part of this was done by ADS, investigating the use of Frama-C to verify if some hypercall implementations satisfied the requirements. That work mainly looked at some hypercall code in isolation and explored what kinds of annotations made sense.

Here we discuss some additional exploration we did that tried to look at how we would go about creating a formal "evidence chain" that would connect the low-level Frama-C annotations up to some formal model of the relevant requirements.

E.1 Relevant Requirements

We identified the following requirments as relevant to this study. Most are focussing on the trap/interrupt setup and handling performed by the kernel.

PK-230 Partitions shall be executed in user mode.

PK-99 The partitioning kernel shall allow a partition to be stopped by itself, by an authorised partition or by the health monitoring.

PK-58 The Partitioning Kernel shall detect and handle all traps and interrupts.

PK-59 Configuration shall allow defining the list of virtual interrupts and traps.

PK-67 Partitioning kernel shall be able to handle multiplexed interrupts and provides associated virtual interrupts to partitions.

PK-60 Configuration shall allow the allocation of a virtual trap to a partition.

PK-61 Configuration shall allow the allocation of a virtual interrupt to a partition.

PK-69 The Partitioning Kernel shall allow a partition to attach a partition handler to each of the virtual traps and interrupts allocated to the said partition.

PK-186 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.

E.2 Selected Hypercalls

Two hypercalls that partitions could use to requet kernel services were chosen by ADS for closer study:

SwitchSchedPlanSys This is a request to change schedule plans. This call will only succeed if the calling partition is authorised to make this call.

SuspendPartitionSys This is a request to suspend an identified partition. This always succeeds if a partition requests its own suspension (a.k.a. "yielding") but only allows authorised partitions to suspend others.

Both are defined in `core/kernel/hypercalls.c`

E.3 Approach

The basic idea was to try and establish how these hypercall were invoked, so as to establish the precise context in which they ran. This would help us identify what pre-conditions would be guaranteed by their callers, which we could then assume as given, in order to prove their behaviour was correct.

What emerged what that there are two distinct calling routes, one used by partitions, and a different one employed by Health Monitoring (HM).

E.4 In Detail: `SwitchSchedPlanSys`

The code for this hypercall is as follows:

```
1  __hypercall xm_s32_t SwitchSchedPlanSys(xm_u32_t newPlanId
2                          , xm_u32_t *__gParam currentPlanId) {
3      localSched_t *sched=GET_LOCAL_SCHED();
4      ASSERT(!HwIsSti());
5      if (!ARE_KTHREAD_FLAGS_SET(sched->cKThread, KTHREAD_SYSTEM_F))
6          return XM_PERM_ERROR;
7      if (CheckGParam(currentPlanId,
8                      sizeof(xm_u32_t), 4, PFLAG_RW|PFLAG_NOT_NULL)<0)
9          return XM_INVALID_PARAM;
10     if (sched->data->cyclic.plan.current->id==newPlanId)
11         return XM_NO_ACTION;
12     if ((newPlanId<1)||
13         (newPlanId>=xmcTab.hpv.cpuTab[GET_CPU_ID()].noSchedCyclicPlans))
14        return XM_INVALID_PARAM;
15     if (SwitchSchedPlan(newPlanId, currentPlanId))
16         return XM_INVALID_CONFIG;
17     return XM_OK;
18 }
```

Essentially we see a number of checks to screen out unauthorised partitions and bad hypercall parameters.

The `CheckGParam` test is coded as follows:

```
static inline xm_s32_t CheckGParam( void *param, xmSize_t size
                                  , xm_u32_t aligment
                                  , xm_s32_t flags)
{
  if (!(flags&PFLAG_NOT_NULL)&&!param)
     return 0;
  if ( ( ((xmAddress_t)param >= CONFIG_XM_OFFSET)
         &&
         ((xmAddress_t)param<(CONFIG_XM_OFFSET+1*16*1024*1024))
       )
       ||
       ( ((xmAddress_t)param+size >= CONFIG_XM_OFFSET)
         &&
         (((xmAddress_t)param+size)<(CONFIG_XM_OFFSET+1*16*1024*1024))
       )
     )
     return -1;
  return 1;
}
```

Then `SwitchSchedPlan` is called to do the work:

```
xm_s32_t SwitchSchedPlan(xm_u32_t newPlanId, xm_u32_t *oldPlanId)
{
  localSched_t *sched=GET_LOCAL_SCHED();
  *oldPlanId=-1;
  if (sched->data->cyclic.plan.current)
     *oldPlanId=sched->data->cyclic.plan.current->id;
  sched->data->cyclic.plan.new
   = &xmcSchedCyclicPlanTab[xmcTab.hpv.cpuTab[GET_CPU_ID()]
       .schedCyclicPlansOffset+newPlanId];
  return 0;
}
```

What is already clear is that we need a lot of information in order to do sensible annotations. For example, what exactly is `CheckGParam` checking?

The above is what the hypercall does. But how is it called?
First we look at `core/include/sparcv8/hypercalls.h`:

```
#define __SWITCH_SCHED_PLAN_NR 27
#define NR_HYPERCALLS 36
#define NR_ASM_HYPERCALLS 10
#define ASM_HYPERCALL_TAB(_ahc) \
    __asm__ (".section .ahypercallstab, \"a\"\n\t" \
        ".align 4\n\t" \
        ".long "#_ahc"\n\t" \
        ".previous\n\t")
```

Here we see the definition of a macro that constructs the corresponding trap table entry. In this case we should not have to have a formal semantics for assembly language because what is being specified is a particular kind of jumptable—we should be able to abstract this away as direct call to the hypercall code.

However in `core/include/hypercalls.h` we find:

```
#define HYPERCALLR_TAB(_hc, _args) \
    __asm__ (".section .hypercallstab, \"a\"\n\t" \
        ".align 4\n\t" \
        ".long "#_hc"\n\t" \
        ".previous\n\t" \
            ".section .hypercallflagstab, \"a\"\n\t" \
        ".long (0x80000000|"#_args")\n\t" \
        ".previous\n\t")

#define HYPERCALL_TAB(_hc, _args) \
    __asm__ (".section .hypercallstab, \"a\"\n\t" \
        ".align 4\n\t" \
        ".long "#_hc"\n\t" \
        ".previous\n\t" \
            ".section .hypercallflagstab, \"a\"\n\t" \
        ".long ("#_args")\n\t" \
        ".previous\n\t")
```

Which one is trhe "right one"?
In file core/kernel/sparcv8/hypercall.c we find:

```
// Hypercall table
HYPERCALL_TAB(MulticallSys, 0); // 0
...
HYPERCALL_TAB(SuspendPartitionSys, 1); // 2
...
HYPERCALLR_TAB(SwitchSchedPlanSys, 2); // 27
...
```

which indicates that we need the second .h file above as it is the only place where HYPERCALLR_TAB is defined.

This only serves to stress the difficulty in working with these sources without guidance as to both what code is relevant, and what the overall design principles are.
From `core/objects/hm.c` we discover that HM calls a hypercall directly, which indicates that this code at least is part of the kernel, and not some authorised partition.

```
xm_s32_t HmRaiseEvent(xmHmLog_t *log) {
...
 switch(partitionTab[partitionId]->ctrl.g->cfg->hmTab[eventId].action) {
 ...
   case XM_HM_AC_SUSPEND:
     ASSERT(partitionId!=XM_HYPERVISOR_ID);
#ifdef CONFIG_OBJ_HM_VERBOSE
     kprintf("[HM] Partition %d suspended\n", partitionId);
#endif
     SUSPEND_PARTITION(partitionId);
     Schedule();
     break;
...
}
```

From one of the examples in the distribution `xal/examples/example.004/partition1.c`, we how a partition invokes a hypercall.

```
    PrintStatus();
```

```
    XM_idle_self();
    retValue= XM_suspend_partition(P2);
    printf( "[P%d] Partition %d %d is suspended\n"
          , XM_PARTITION_SELF, P2, retValue);
    if (retValue >=0) PrintStatus();
    XM_idle_self();
    retValue= XM_suspend_partition(P3);
    printf( "[P%d] Partition %d %d is suspended\n"
          , XM_PARTITION_SELF, P3, retValue);
    if (retValue >=0) PrintStatus();
```

It calls XM_suspend_partition defined in `user/libxm/include/xmhypercalls.h`:

```
extern __stdcall xm_s32_t XM_suspend_partition(xm_u32_t partition_id);
extern __stdcall xm_s32_t
   XM_switch_sched_plan( xm_u32_t newPlanId
                       , xm_u32_t *currentPlanId);
```

This is the only place in all the sources that these identifiers occur! We used the "ack" source code search utility to locate identifiers , but failed to find any other textual occurences of these. We cannot find where these are defined, and suspect there may be cpp #define macro hacking being done.
We find the following in `user/libxm/sparcv8/hypercalls.h`:

```
xm_hcall1r(suspend_partition, xm_u32_t, partitionId);
xm_hcall2r( switch_sched_plan, xm_u32_t
          , newPlanId, xm_u32_t *, currentPlanId);
```

and eventually find that these are defined as assembler macros in `user/libxm/include/sparcv8/xmhypercalls.h`:

```
#define _XM_HCALL1(a0, _hc_nr, _r) \
  __asm__ __volatile__ \
   ( "mov %0, %%o0\n\t" \
    "mov %2, %%o1\n\t" \
    __DO_XMHC \
    "mov %%o0, %0\n\t" : "=r" (_r) : "0" (_hc_nr), "r" (a0) : \
    "o0", "o1", "o2", "o3", "o4", "o5",   "o7")

#define xm_hcall1r(_hc, _t0, _a0) \
ASMLINK xm_s32_t XM_##_hc(_t0 _a0) { \
    xm_s32_t _r ; \
    _XM_HCALL1(_a0, _hc##_nr, _r); \
    return _r; \
}
```

Here we need to understand the c calling convention in use here.

E.5 Summary

Our attempt to establish a formal chain was not successful, because our attempt to identify the call chains from partitions to hypercalls took a lot of code searching, and even then resulting in gaps we could not fill. This brings home the point, also made by ADS in their Frama-C investigations, that a high degree of familiarity with the code and the Makefiles that compile the code is needed in order to navigate it to identify how it all connects.

It really does require a full-time commitment by one of more individuals to do this task effectively.

F Frama-C Case-Study Log

F.1 Activity Log

Done

- Setup c-code, `HyperCall.lhs` and this log in Frama-C subdirectory.
- Review "fingered" requirements: 58–61,67, 69, 99, 186 and 230
- Examine how target C-function is situated
- Installing Frama-C
- Identify routes to call each hypercall, noting that HM code seems to do it differently to "normal" Partition code

In Progress

- Informally annotate source code
- ...

Planned

- ...

F.2 Relevant Requirements

Relevant requirements?:

PK-230 Partitions shall be executed in user mode.

PK-99 The partitioning kernel shall allow a partition to be stopped by itself, by an authorised partition or by the health monitoring.

PK-58 The Partitioning Kernel shall detect and handle all traps and interrupts.

PK-59 Configuration shall allow defining the list of virtual interrupts and traps.

PK-67 Partitioning kernel shall be able to handle multiplexed interrupts and provides associated virtual interrupts to partitions.

PK-60 Configuration shall allow the allocation of a virtual trap to a partition.

PK-61 Configuration shall allow the allocation of a virtual interrupt to a partition.

PK-69 The Partitioning Kernel shall allow a partition to attach a partition handler to each of the virtual traps and interrupts allocated to the said partition.

PK-186 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.

F.3 The Hypercalls

Mentioned by Alexandre Cortier (Airbus): `SuspendPartitionSys` and `SwitchSchedPlanSys`.

Name	File	Comment
SwitchSchedPlanSys	core/kernel/hypercalls.c	
SuspendPartitionSys	core/kernel/hypercalls.c	
schedData	core/include/sched.h	
localSched_t	core/include/sched.h	

F.4 In Detail: `SwitchSchedPlanSys`

We present the code as is, and with commentary actions
Code:

It calls the following, which clearly is intended to contain relevant scheduler data.

```
extern localSched_t localSchedInfo[1];
```

whose associated type declarations are the following:

F.4.1 From `/core/include/sched.h`

```
struct schedData {
    struct cyclicData {
        kTimer_t kTimer;
        struct {
            const struct xmcSchedCyclicPlan *current;
            const struct xmcSchedCyclicPlan *new;
            const struct xmcSchedCyclicPlan *prev;
        } plan;
        xm_s32_t slot; // next slot to be processed
        xmTime_t mjf;
        xmTime_t sExec;
        xmTime_t planSwitchTime;
        xmTime_t nextAct;
        kThread_t *kThread;
    } cyclic;
};

typedef struct {
    kThread_t *idleKThread;
    kThread_t *cKThread;
    kThread_t *fpuOwner;
    kThread_t *schedThreadTab;
    struct schedData *data;
    xm_u32_t flags;
#define LOCAL_SCHED_ENABLED 0x1
} localSched_t;

extern localSched_t localSchedInfo[1];

#define GET_LOCAL_SCHED() localSchedInfo
```

We immediately note that the following are all the same (at least for sparcv8 architectures):

`xm_u32_t xmAddress_t xmIoAddress_t xmSize_t xmId_t`

Here is the `kThread` stuff:

```
typedef struct kThread {
   // Harcoded, don't change it
   xm_u32_t magic1;
   // Harcoded, don't change it
   xmAddress_t *kStack;
   volatile xm_u32_t flags;
   //  [3...0] -> scheduling bits
   struct dynList localActiveKTimers;
   struct guest *g;
   void *schedData;
       cpuCtxt_t *irqCpuCtxt;
   xm_u32_t irqMask;
   xm_u32_t magic2;
   } ctrl;
typedef xm_u8_t kThread_t[8192];
```

It also call the function below which is clearly a 3-valued predicate $(-1, 0, 1)$ or status function.

```
/*@
  @ requires \valid((xm_u32_t*)param);
  @ ensures \result == 1;
  */
static inline xm_s32_t CheckGParam( void *param, xmSize_t size
                                , xm_u32_t aligment
                                , xm_s32_t flags)
{
  if (!(flags&PFLAG_NOT_NULL)&&!param)
     return 0;
  if ( ( ((xmAddress_t)param >= CONFIG_XM_OFFSET)
         &&
         ((xmAddress_t)param<(CONFIG_XM_OFFSET+1*16*1024*1024))
       )
       ||
       ( ((xmAddress_t)param+size >= CONFIG_XM_OFFSET)
         &&
         (((xmAddress_t)param+size)<(CONFIG_XM_OFFSET+1*16*1024*1024))
       )
     )
     return -1;
  return 1;
}
```

Here is the code that does the real work!

```
xm_s32_t SwitchSchedPlan(xm_u32_t newPlanId, xm_u32_t *oldPlanId)
{
  localSched_t *sched=GET_LOCAL_SCHED();

  *oldPlanId=-1;
  if (sched->data->cyclic.plan.current)
     *oldPlanId=sched->data->cyclic.plan.current->id;

  sched->data->cyclic.plan.new
   = &xmcSchedCyclicPlanTab[xmcTab.hpv.cpuTab[GET_CPU_ID()]
       .schedCyclicPlansOffset+newPlanId];

  return 0;
}
```

F.5 Bits ...

F.5.1 From core/include/sparcv8/hypercalls.h

```
#define __SUSPEND_PARTITION_NR 2

#define NR_HYPERCALLS 36


//@ \void{<track id="asm-hypercall-numbers">}
#define sparc_iret_nr 0
#define sparc_flush_regwin_nr 1
#define sparc_get_psr_nr 2
#define sparc_set_psr_nr 3
#define sparc_set_pil_nr 4
#define sparc_clear_pil_nr 5
#define sparc_ctrl_winflow_nr 6
#define sparc_set_irqmask_nr 7
#define sparc_clear_irqmask_nr 8
#define sparc_clear_irqs_nr 9
//@ \void{</track id="asm-hypercall-numbers">}

#define NR_ASM_HYPERCALLS 10

#define ASM_HYPERCALL_TAB(_ahc) \
    __asm__ (".section .ahypercallstab, \"a\"\n\t" \
        ".align 4\n\t" \
        ".long "#_ahc"\n\t" \
        ".previous\n\t")
```

F.5.2 From core/kernel/sparcv8/hypercalls.c

```
HYPERCALL_TAB(SuspendPartitionSys, 1); // 2

HYPERCALLR_TAB(SwitchSchedPlanSys, 2); // 27

// ASM hypercall table
ASM_HYPERCALL_TAB(SparcIRetSys);
ASM_HYPERCALL_TAB(SparcFlushRegWinSys);
ASM_HYPERCALL_TAB(SparcGetPsrSys);
ASM_HYPERCALL_TAB(SparcSetPsrSys);
ASM_HYPERCALL_TAB(SparcSetPilSys);
ASM_HYPERCALL_TAB(SparcClearPilSys);
ASM_HYPERCALL_TAB(SparcCtrlWinFlowSys);
ASM_HYPERCALL_TAB(SparcSetIrqMask);
ASM_HYPERCALL_TAB(SparcClearIrqMask);
ASM_HYPERCALL_TAB(SparcClearIrqsSys);
```

F.5.3 From core/kernel/hypercalls.c

```
xm_s32_t SwitchSchedPlanSys( xm_u32_t newPlanId
                          , xm_u32_t * currentPlanId)
{
  localSched_t *sched=localSchedInfo;
  if ((*(sched->cKThread->ctrl)).flags&(KTHREAD_SYSTEM_F))
     return XM_PERM_ERROR;
  if (CheckGParam( currentPlanId, sizeof(xm_u32_t)
                , 4, PFLAG_RW|PFLAG_NOT_NULL ) <0 )
     return XM_INVALID_PARAM;
  if (sched->data->cyclic.plan.current->id==newPlanId)
     return XM_NO_ACTION;
  if ( (newPlanId<1) ||
       (newPlanId>=xmcTab.hpv.cpuTab[GET_CPU_ID()].noSchedCyclicPlans))
     return XM_INVALID_PARAM;
  if (SwitchSchedPlan(newPlanId, currentPlanId))
     return XM_INVALID_CONFIG;
  return XM_OK;
}

__hypercall xm_s32_t SuspendPartitionSys(xmId_t partitionId) {
    localSched_t *sched=GET_LOCAL_SCHED();

    ASSERT(!HwIsSti());
    if (partitionId!=sched->cKThread->ctrl.g->cfg->id) {
   if (!ARE_KTHREAD_FLAGS_SET(sched->cKThread, KTHREAD_SYSTEM_F))
       return XM_PERM_ERROR;
   if ((partitionId>=xmcTab.noPartitions))
       return XM_INVALID_PARAM;
        else if (!partitionTab[partitionId])
       return XM_INVALID_PARAM;

        if ( ARE_KTHREAD_FLAGS_SET(partitionTab[partitionId]
           , KTHREAD_HALTED_F)==KTHREAD_HALTED_F)
       return XM_OP_NOT_ALLOWED;

        SUSPEND_PARTITION(partitionId);
   return XM_OK;
    }

    SUSPEND_PARTITION(partitionId);
    Schedule();
    return XM_OK;
}
```

F.5.4 From core/objects/hm.c

```
xm_s32_t HmRaiseEvent(xmHmLog_t *log) {
...
 switch(partitionTab[partitionId]->ctrl.g->cfg->hmTab[eventId].action) {
 ...
   case XM_HM_AC_SUSPEND:
     ASSERT(partitionId!=XM_HYPERVISOR_ID);
```

```
#ifdef CONFIG_OBJ_HM_VERBOSE
     kprintf("[HM] Partition %d suspended\n", partitionId);
#endif
     SUSPEND_PARTITION(partitionId);
     Schedule();
     break;
...
}
```

F.5.5 From core/objects/hmapp.c

```
xm_s32_t HmAppRaiseEvent(xmHmAppLog_t *log) {
...
 switch(partitionTab[partitionId]->ctrl.g->cfg->hmTab[eventId].action) {
...
   case XM_HM_AC_SUSPEND:
     ASSERT(partitionId!=XM_HYPERVISOR_ID);
#ifdef CONFIG_OBJ_HM_VERBOSE
     kprintf("[HM] Partition %d suspended\n", partitionId);
#endif
     SUSPEND_PARTITION(partitionId);
     Schedule();
     break;
..
}
```

F.5.6 From xal/examples/example.004/partition1.c

```
    PrintStatus();
    XM_idle_self();
    retValue= XM_suspend_partition(P2);
    printf( "[P%d] Partition %d %d is suspended\n"
          , XM_PARTITION_SELF, P2, retValue);
    if (retValue >=0) PrintStatus();
    XM_idle_self();
    retValue= XM_suspend_partition(P3);
    printf( "[P%d] Partition %d %d is suspended\n"
          , XM_PARTITION_SELF, P3, retValue);
    if (retValue >=0) PrintStatus();
```

F.5.7 From user/libxm/include/xmhypercalls.h

This is the only place in all the sources that these identifiers occur!

```
// Partition status hypercalls
extern __stdcall xm_s32_t XM_suspend_partition(xm_u32_t partition_id);

extern __stdcall xm_s32_t
   XM_switch_sched_plan( xm_u32_t newPlanId
                       , xm_u32_t *currentPlanId);
```

F.5.8 From user/libxm/sparcv8/hypercalls.h

```
xm_hcall1r(suspend_partition, xm_u32_t, partitionId);

xm_hcall2r( switch_sched_plan, xm_u32_t
          , newPlanId, xm_u32_t *, currentPlanId);
```

F.5.9 From core/include/sparcv8/linkage.h

```
#define ALIGNMENT 8
#define ASM_ALIGN .align ALIGNMENT
#define __stdcall
#define __hypercall __attribute__((noinline, section(".text")))
#define ALIGNED_C __attribute__((aligned(8)))
```

F.5.10 From core/include/hypercalls.h

```
#define HYPERCALLR_TAB(_hc, _args) \
    __asm__ (".section .hypercallstab, \"a\"\n\t" \
        ".align 4\n\t" \
        ".long "#_hc"\n\t" \
        ".previous\n\t" \
            ".section .hypercallflagstab, \"a\"\n\t" \
        ".long (0x80000000|"#_args")\n\t" \
        ".previous\n\t")

#define HYPERCALL_TAB(_hc, _args) \
    __asm__ (".section .hypercallstab, \"a\"\n\t" \
        ".align 4\n\t" \
        ".long "#_hc"\n\t" \
        ".previous\n\t" \
            ".section .hypercallflagstab, \"a\"\n\t" \
        ".long ("#_args")\n\t" \
        ".previous\n\t")
```

F.5.11 From core/include/sched.h

```
static inline void SUSPEND_PARTITION(xmId_t id) {
    SET_KTHREAD_FLAGS(partitionTab[id], KTHREAD_SUSPENDED_F);
}
```

F.5.12 From `user/libxm/include/sparcv8/xmhypercalls.h`

```
#define __STR(x) #x
#define TO_STR(x) __STR(x)
#define ASMLINK

/* <track id="DOC_HYPERCALL_MECHANISM"> */
#define __DO_XMHC "ta "TO_STR(XM_HYPERCALL_TRAP)"\n\t"
/* </track id="DOC_HYPERCALL_MECHANISM"> */

#define _XM_HCALL0(_hc_nr, _r) \
 __asm__ __volatile__ \
  ( "mov %0, %%o0\n\t" \
   __DO_XMHC \
    "mov %%o0, %0\n\t" : "=r" (_r) : "0" (_hc_nr) : \
    "o0", "o1", "o2", "o3", "o4", "o5",   "o7")

#define _XM_HCALL1(a0, _hc_nr, _r) \
  __asm__ __volatile__ \
   ( "mov %0, %%o0\n\t" \
    "mov %2, %%o1\n\t" \
    __DO_XMHC \
    "mov %%o0, %0\n\t" : "=r" (_r) : "0" (_hc_nr), "r" (a0) : \
    "o0", "o1", "o2", "o3", "o4", "o5",   "o7")

#define _XM_HCALL2(a0, a1, _hc_nr, _r) \
__asm__ __volatile__ \
 ( "mov %0, %%o0\n\t" \
   "mov %2, %%o1\n\t" \
   "mov %3, %%o2\n\t" \
   __DO_XMHC \
   "mov %%o0, %0\n\t" : "=r" (_r) : "0" (_hc_nr), "r" (a0), "r" (a1) : \
   "o0", "o1", "o2", "o3", "o4", "o5",   "o7")
```

```
#define xm_hcall0(_hc) \
ASMLINK void XM_##_hc(void) { \
    xm_s32_t _r ; \
    _XM_HCALL0(_hc##_nr, _r); \
}

#define xm_hcall0r(_hc) \
ASMLINK xm_s32_t XM_##_hc(void) { \
    xm_s32_t _r ; \
    _XM_HCALL0(_hc##_nr, _r); \
    return _r; \
}

#define xm_hcall1r(_hc, _t0, _a0) \
ASMLINK xm_s32_t XM_##_hc(_t0 _a0) { \
    xm_s32_t _r ; \
    _XM_HCALL1(_a0, _hc##_nr, _r); \
    return _r; \
}
#define xm_hcall2r(_hc, _t0, _a0, _t1, _a1) \
ASMLINK xm_s32_t XM_##_hc(_t0 _a0, _t1 _a1) { \
    xm_s32_t _r ; \
    _XM_HCALL2(_a0, _a1, _hc##_nr, _r); \
    return _r; \
}

/* <track id="DOC_HYPERCALL_MECHANISM"> */
#define __XM_HC ta XM_HYPERCALL_TRAP ; nop
/* </track id="DOC_HYPERCALL_MECHANISM"> */
#define __XM_AHC ta XM_ASMHYPERCALL_TRAP ; nop
```

F.5.13 From core/include/kthread.h

```
static inline void SET_KTHREAD_FLAGS(kThread_t * k,int f){
  do{
     // ASSERT(
      //  !(k)->ctrl.g ||
          !(k)->ctrl.g->partCtrlTab ||
          (((k)->ctrl.flags&
             ~(KTHREAD_FLUSH_CACHE_W<<KTHREAD_FLUSH_CACHE_B))
           ==
           ((k)->ctrl.g->partCtrlTab->flags&
             ~(KTHREAD_FLUSH_CACHE_W<<KTHREAD_FLUSH_CACHE_B)))
      // );

     (*(ctrl *)k).flags|=(f);
    if ((*(ctrl *)k).g&&(*(ctrl *)k).g->partCtrlTab)
       (*(ctrl *)k).g->partCtrlTab->flags|=(f);
  } while(0);
}
```

G Issues

Here we look at issues with the Requirements Baseline [2].

The observation section contains general unclassifed comments, which may be ephemeral. The remaining three sections list issues classified as Major, Minor or Editorial as per ESA RIDs.

G.1 Observations

The need for invariants abound. For example,ports have max message sizes (**PK-3**2), so we need to ensure the ports on a given channel have the same max!

"communication" used as an alias for "channel" (**PK-4**1,**PK-4**6)

If HM configuration says HM event e is ignored (**PK-9**2,**PK-5**58), then does **PK-9**8 still require the ignored event to be logged (i.e., is ignoring an event also considered a way of "handling" it?)

The LEON3-FT produces an interrupt if the memory controller detects a 2-bit or worse error during a memory read. How does this fit in with the requirements for FDIR?

Configuration data is static, *not part of the kernel* (?), but cannot be accessed directly by the partitions (so the kernel needs access, and to protect it) [RB §3.8, p31].

G.2 Major

- As pointed out in Sect.**??**, the requirements document is a bit vague concerning chanels and ports. We shouldn't have had to look at XtratuM config files in order to figure out what was intended.
- c As is made very clear in both Sect.D and Sect.E, there really needs to much more guidance as to the structure and architecture of the code and its build system. It is very difficult to find what code is actually part of a running kernel.
- As a follow-on from the previous point: while it may be feasible to attempt a formal verification in close co-operation with the code vendors, it is currently hard to see how an independent 3rd-party could do this.

G.3 Minor

None at present

G.4 Editorial

- **PK-8**1 refers to a non-existent **PK-1**53 in its Note.
- **PK-4**68 refers to a non-existent **PK-4**23 in its Note.
- In **PK-1**97, in what units is the size of the MAF measured?
- In the parameter table on p92, various durations and times are given, but no units are supplied.
- **PK-9**7 and **PK-9**8 are closely related but their rationales are almost identical.

References

[1] Kevin Hennessy. Investigating Interrupts in the Xtratum Hypervisor using CSP. Master in computer science dissertation, Computer Science and Statistics, Trinity College Dublin, May 2016.

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

[3] A. Butterfield. Formal Modelling Workshop (I). Powerpoint, presented at IMA-KQP Core Requirements Workshop, October 2014. ESTEC Contract No. 4000112208/14/NL/GLC/.

[4] C. A. R. Hoare. *Communicating Sequential Processes*. Prentice-Hall, Englewood Cliffs, New Jersey, 1985.

[5] Loïc Correnson et al. *Frama-C User Manual*. CEA List, Software Safety Laboratory, Saclay, F-91191, release neon-20140301 edition, 2014. `http://frama-c.com/download/frama-c-user-manual.pdf`.

[6] A. Butterfield. DD1: MTOBSE workshop materials. Three powerpoint presentations, for each IMA-KQP Workshop, 2014-15. ESTEC Contract No. 4000112208/14/NL/GLC/.

[7] A. Butterfield. DD2: Input material to IMA-KP SRR document D01, "consolidated input documentation for IMA Separation Kernel requirements". Technical Report Version 1.0, Trinity College, the University of Dublin, 2015. ESTEC Contract No. 4000112208/14/NL/GLC/.

[8] A. Butterfield. DD3: Input material to IMA-KP SRR document D02, "IMA Separation Kernel requirements baseline". Technical Report Version 1.0, Trinity College, the University of Dublin, 2015. ESTEC Contract No. 4000112208/14/NL/GLC/.

[9] A. Butterfield. Formal Modelling Workshop (II). Powerpoint, presented at IMA-KQP Extended Requirements Workshop, October 2014. ESTEC Contract No. 4000112208/14/NL/GLC/.

[10] A. Butterfield. Formal Modelling Workshop (III). Powerpoint, presented at IMA-KQP Qualification Methods and Techniques Workshop, October 2014. ESTEC Contract No. 4000112208/14/NL/GLC/.

[11] A. Butterfield and M. Hinchey. Towards the adoption of formal techniques for kernel qualification. In *DASIA 2015, The International Space System Engineering Conference*, Barcelona, Spain, May 2015. ASD-Eurospace.

[12] A. Butterfield. DD4: Input material to IMA-KP TRR document D06, "Strategy and organisation for qualification of IMA Separation Kernels". Technical Report Version 1.0, Trinity College, the University of Dublin, 2015. ESTEC Contract No. 4000112208/14/NL/GLC/.

[13] A. Butterfield. DD5: Input material to IMA-KP TRR document D08, "Methods and techniques for qualification of IMA Separation Kernels". Technical Report Version 1.0, Trinity College, the University of Dublin, 2015. ESTEC Contract No. 4000112208/14/NL/GLC/.

[14] Aeroflex Gaisler. *LEON3-FT SPARC V8 Processor LEON3FT-RAX Data Sheet and User's Manual*. Aeroflex Gailser, version 1.9 edition, January 2013.

[15] Thomas Gibson-Robinson, Philip Armstrong, Alexandre Boulgakov, and A.W. Roscoe. FDR3 — A Modern Refinement Checker for CSP. In Erika brahm and Klaus Havelund, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, volume 8413 of *Lecture Notes in Computer Science*, pages 187–201, 2014.

[16] Thomas Gibson-Robinson, Philip Armstrong, Alexandre Boulgakov, and A.W. Roscoe. *Failures Divergences Refinement (FDR) Version 3*, 2013.

[17] Alexandre Cortier. D23: Guidelines on formal methods environment and analysis of formal methods technique, institution = AIRBUS Defence and Space SAS, year = 2016, number = Issue 1.1, note = ESTEC Contract No. 4000111495/14/NL/GLC/al,. Technical report.

[18] A. Butterfield. DD6: Input material to IMA-KP QR document D11, "Evaluation of methods and techniques for qualification of IMA Separation Kernels". Technical Report Version 1.0, Trinity College, the University of Dublin, 2015. ESTEC Contract No. 4000112208/14/NL/GLC/.

[19] A. Butterfield, Alexandre Cortier, Kevin Hennessy, and M. Hinchey. Towards formal verification of interrupts and hypercalls. In *DASIA 2015, The International Space System Engineering Conference*, Tallinn, Estonai, May 2015. ASD-Eurospace.

[20] A. Butterfield. Formal methods expert to ima-sp kernel qualification - preparation. Powerpoint, presented at TEC-SW Final Presentation Days, ESTEC, 9th-10th June, June 2016. ESTEC Contract No. 4000112208/14/NL/GLC/.

[21] A. Butterfield. DD7: Input material to IMA-KP AR document D14, "Final Report". Technical Report Version 1.0, Trinity College, the University of Dublin, 2015. ESTEC Contract No. 4000112208/14/NL/GLC/.

[22] A. Butterfield. R1: Formal Methods Expertise — Final Report. Technical Report Version 1.0, Trinity College, the University of Dublin, 2016. ESTEC Contract No. 4000112208/14/NL/GLC/.

[23] Gerwin Klein, June Andronick, Kevin Elphinstone, Gernot Heiser, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, Thomas Sewell, Harvey Tuch, and Simon Winwood. seL4: Formal verification of an OS kernel. *Communications of the ACM (CACM)*, 53(6):107–115, June 2010.

[24] Thierry Lecomte. Applying a formal method in industry: A 15-year trajectory. In María Alpuente, Byron Cook, and Christophe Joubert, editors, *FMICS*, volume 5825 of *Lecture Notes in Computer Science*, pages 26–34. Springer, 2009.

[25] Thierry Lecomte. Atelier-B has turned Twenty. Invited Talk, ABZ2016, Linz, Austria, 2016.