TOWARDS FORMAL VERIFICATION OF INTERRUPTS AND HYPERCALLS

Andrew Butterfield[1], **Alexandre Cortier**[2], **Kevin Hennessy**[3], **and Mike Hinchey**[4]

[1]*Lero, Trinity College Dublin,Andrew.Butterfield@scss.tcd.ie*
[2]*Airbus Defence and Space*
[3]*Trinity College Dublin*
[4]*Lero, University of Limerick*

1. INTRODUCTION

The project *IMA-SP Kernel Qualification Preparation (IMAKQP)* is an ongoing activity led by SciSys Ltd, involving CNES, Airbus DS, TASF, looking at the approach to be adopted for IMA-SP Separation Kernel qualification. The activity *Formal Methods Expert to IMA-SP Kernel Qualification Preparation (FMEIMAKQP)* [1] is a parallel joint activity involving Lero@TCD, exploring the potential role that formal verification techniques might play in a future kernel qualification process. It follows on from the experience of a previous activity *Method and Tools for Onboard Software Engineering (MTOBSE)* [2]

The MTOBSE project, involving two partners in Lero: the Irish Software Research Centre, namely the University of Limerick and Trinity College Dublin, looked at a top-down approach to formalising and verifying an IMA TSP kernel, and was reported on at two previous DASIA events[BSH13, BSH13]. We developed a reference specification for a Time-Space Partitioning (TSP) kernel[BS13a] and did a survey of formal techniques that had potential for verifying a kernel implementation[BS13b, BS13c]. We then selected Isabelle/HOL[NPW02] and used this to formalise the reference specification and some of the XtratuM code[BS13d]. We then reported on our experiences[BS13e]. A key focus of the MTOBSE project was to consider formally verifying a kernel to the extent required by the Common Criterion (CC) guidelines for both functional correctness and security[Cri05], with particular reference to the Separation Kernel Protection Profile (SKPP)[Dir07].

The current focus of FMEIMAKQP is exploring approaches for qualifying an *existing* separation kernel implementation as its starting point. In particular, the focus is on functional correctness rather than security properties. From a formal perspective this involves looking much more closely at techniques for reasoning at or close to source code level. Also of interest is to explore how formal techniques might best be integrated with well-established software engineering practises, such as testing, and model-based development. Preliminary steps at exploring these aspects was reported at the most recent DASIA[BH15]. The relevant approaches identified then included:

- code annotation of pre- and post-conditions with a back-propagation algorithm that could automatically generate small, often simple verification conditions, mostly amenable to automatic proof tools [CDH$^+$09, C$^+$15].
- Many formal models can be used to construct graph-like structures known as labelled transition graphs, and these can be used as a basis for verification techniques known as model-based testing[Tre08]. An extreme form of this is model-checking, where an exhaustive search over the entire graph is used to verify or refute properties stated in a temporal logic notation [QS82, CES86].
- A key component of any deployed kernel is its configuration, as defined by the system integrator. A formal model of the configuration defines the datatypes and relationships involved and an invariant property that characterises when such a configuration is well-formed. In particular, the invariant describes properties that will be required by other verification techniques.

Here we describe progress made so far in this investigation, where a decision has been made to limit the scope to aspects of the kernel behaviour and requirements that are centered around interrupt handling. This is done for two reasons: (i) the kernel as a whole is quite complex and this activity is an exploratory one, so diving deep into a few aspects was considered a more productive way to get useful information and experience; and (ii) the behaviour, timing and unpredictability of interrupts make testing them quite difficult, and so it looks like a good candidate for exploring formal techniques. We will discuss progress focussing on two case-studies:

[1] funded by ESTEC CONTRACT No. 4000112208 supported, in part, by Science Foundation Ireland grant 13/RC/2094

[2] funded by ESTEC CONTRACT No. 4000106016, and supported, in part, by Science Foundation Ireland grant 10/CE/I1855

- using the CSP process algebra[Hoa85] and its associated model/refinement checking tool FDR3[GRABR14] to model the hardware platform and kernel/partition runtime behaviour.
- using the Frama-C tool[C+15] to explore C-code annotations for verifying selected hypercalls.

2. UNAVOIDABLE CONCURRENCY

The purpose of the separation kernel is to allow multiple (software) partitions to run on one hardware platform, each being isolated in both time and space, with any cross-communication limited to what is authorised. Partitions will have varying degrees of privilege and access depending on their function and their degree of criticality and trustworthiness.

The role played by the kernel is to orchestrate the execution of the partitions according to a configuration (schedules and access rights) determined by the system integrator. The kernel does this by ensuring that key "oversight" hardware, e.g. the memory management unit (MMU), or memory protection unit (MPU), is properly configured, and then using timers and interrupts to manage the scheduling and context-switches.

For a single-core system, despite the fact that at any point in time, we have only one partition or the kernel itself actually running, we find that we have what is an essentially *concurrent* system. The CPU executes instruction in a sequential manner on behalf of either the kernel or partition, which generates traffic on the platforms address/data/signalling busses. The memory (MEM) runs in parallel responding to CPU memory requests, but this is not the parallel/concurrent behaviour that concerns us. Also running concurrently is the MMU/MPU which observes the bus traffic, and checks, against its current configuration setup, in the memory requests are permitted. If not, it then raises a high-priority memory fault interrupt. Meanwhile, the interrupt request hardware (IRQ), although part of the CPU, is effectively also running in parallel, taking in interrupt requests from IO devices and timers (DEV), as well as the MMU/MPU. Again based on its configuration, it decides if and when to signal an interrupt to the CPU. In effect we have a concurrent system at work, a lot of whose behaviour comes from appropriately configured hardware (Fig. 1) . This hardware is configured by the kernel software, and needs to be revised by that software every time there is a context switch between the kernel and/or partitions.

We have chosen to construct models of both the required kernel/partition behaviour and the target hardware platform used for implementation. Because of the inherently concurrent nature of the computing environment in which the kernel must operate, we have chosen to use a formalism designed for concurrency, namely Communicating Sequential Processes (CSP)[Hoa85], for its suitability, and the availability of its powerful analysis tool

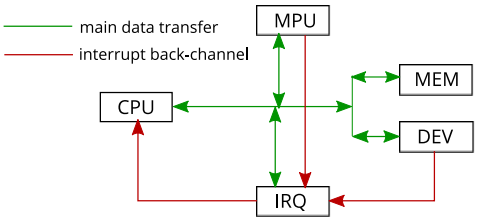

Figure 1. Concurrent Hardware Platform

FDR3 [GRABR14]. The plan is to model both the kernel/partition required behaviour, and the hardware platform concurrent behaviour in CSP, and the use FDR3 to assess how well they conform. A key plan here is that developing these models will give insights into how to best approach the formal verification of the interrupt setup and handling code.

We are currently building the hardware platform CSP model incrementally, starting with the CPU/MEM interaction, then adding in the MMU/MPU component, initially just signalling any detected memory faults.

```
datatype ADDRESS = a0 | a1
datatype DATA    = d0 | d1
datatype DIR     =  rd | wr

channel
  transfer : DIR.ADDRESS -- address, transfer direction by CPU
  write    : DATA        -- data write by CPU
  read     : DATA        -- data read by CPU

CPU =  CPUFETCH
CPUFETCH = |~| a : ADDRESS @ transfer.rd.a -> read?d -> CPUEXECUTE
CPUEXECUTE = ( CPUREAD |~| CPUWRITE |~| CPU)
CPUREAD =  |~| a : ADDRESS @ transfer.rd.a -> read?d -> CPU
CPUWRITE = |~| a : ADDRESS, d : DATA @ transfer.wr.a -> write.d -> CPU


mem_initial = (| a0 => d0, a1 => d0 |)
MEM(m) = MEMREAD(m) [] MEMWRITE(m)
MEMREAD(m) = transfer.rd?a -> MEMREADDATA(m,a)
MEMREADDATA(m,a) = read.(mapLookup(m,a)) -> MEM(m)
MEMWRITE(m) =  transfer.wr?a -> MEMWRITEDATA(m, a)
MEMWRITEDATA(m, a) = write?d -> MEM(mapUpdate(m, a, d) )

CPU_MEM = CPU [| {|transfer,read,write|} |] MEM(mem_initial)


channel memfault : DIR.ADDRESS

mmu_forbidden= {wr.a0, rd.a0}

MMU(mmu_forbidden) = transfer?dir?a ->  if member(dir.a, mmu_forbidden)
                                        then MEMFAULT(dir, a)
                                        else MMU(mmu_forbidden)
MEMFAULT(dir, a) = memfault.dir.a -> MMU(mmu_forbidden)

SYSTEM = CPU_MEM [| {|transfer |} |] MMU(mmu_forbidden)
```

The the IRQ component will be added to complete the loop back to the CPU, whose behaviour will then be extended to perform checks for interrupts at the appropriate times, and to model the switch between user and supervisor modes. This will result in a model that capture the "hardware view" of what is taking place in the system.

A separate model will talk about memory accesses, interrupts from the perspective of entities such as the kernel or a given partition. This will, in effect, define a "kernel implementation view" of the system behaviour. We should be able to show that its behaviour is a subset of that of the hardware view — in other words that it is consistent with what the hardware will allow.

We can then use CSP to construct small models that in

effect act as requirement tests, which can the be used to check the implementation correctness.

3. HYPERCALLS: THE DETAIL IS THE DEVIL

The relatively abstract CSP models of platform and kernel concurrency mainly serve to help understand the functional relationships that hold between the various fragments of C code and assembler that make up the actual kernel code that we are trying to verify formally. We are exploring the use of the Frama-C toolkit [C+15] to do this verification, and are focussing in on two hypercalls in the XtratuM sources.

Hypercalls are the mechanism used by partitions to call for kernel services, such as channel communication, virtual interrupt setup, changing schedule plans, signalling early completion, and so on. They are typically invoked using a TRAP instruction, which results in the appropriate kernel-installed handler being executed, which then checks for the appropriate permission before delivering the requested service. We have focussed on two hypercalls:

- *Suspend Partition* can be invoked by any partition to signal its own completion to the kernel, or be used by a privileged partion (usually health monitoring) to stop some other partition. Main entry point: `SuspendPartitionSys`
- *Switch Schedule Plan* is used by an authorised partition to changeover from the current partition schedule to a new one. This is typically done as a mission enters a new phase that requires the software load and configuration to change. Main entry point: `SwitchSchedPlanSys`

The challenge here is to identify: (i) what these hypercalls actually do, (ii) what data they need to perform their task and (iii) where and how they actually get invoked. The main issue we currently are encountering is that we really need clear design documentation that describes the architecture, data type design and functional call graphs of the code, as well as access to the kernel developers for their expertise regarding the code and its interdependencies. Not having these does place liits on how far we can currently experiment.

Some experimentation with Frama-C has allowed us to uncover new information about hidden pre-conditions. Consider the following function `CheckGParam` that checks various aspects of parameters sent to `SwitchSchedPlanSys`:

```
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;
}
```

The corresponding Frama-C annotation, defining pre-conditions (`requires`,`assumes`) and post-conditions (`ensures`) is largely just an appropriate rewriting of the if-conditions, except for the following pre-condition fragment, whose inclusion was required to get the tool to accept the code as correct.

```
requires sizeof(param)==sizeof(xm_u32_t);
```

This has exposed an unstated assumption about data sizes.

The challenge with using Frama-C code annotations as the basis for proving kernel correctness is that we need to formalise all the data-invariants that characterise how the kernel configuration data gets reified as concrete C-datastructure values. It is here where good design documentation and/or access to expertise would be most helpful.

4. CONCLUSIONS AND FUTURE WORK

FMEIMAKQP was a small consultative and exploratory activity, that explored various possible approaches to using formal techniques to assist in kernel qualification. It is quite clear that high-level formal modelling helps to "frame" the verification task, by exposing essential concurrency, while avoiding that which is irrelevant. High level models can also be used to check completness and consistency of the requirements baseline.

Low level (code) formal annotations can also work, but the connection to the top-level (requirments) is decidely non-trivial, as we need a "semantic bridge" between C data and kernel concepts. This bridge can only be succesfully developed in close collaboration with the kernel developers.

One of the requirements we did not look at from a formal perspective was PK-9, which is the requirement that partition contexts are saved and restored properly during partition context switches. This requirement proved particularly hard to test, and now looks like an interesting challenge candidate for a possible future investigation into the top-to-toe verification of a single requirement.

Possible next steps include:

- Formalising the Requirements Baseline
- Doing a "deep-dive" to formally link a requirement (e.g. PK-9) to its code.
- Applying formal techniques in other space-software domains, particularly those currently in development using model-based techniques.

REFERENCES

[BH15] A. Butterfield and M. Hinchey. Towards the adoption of formal techniques for kernel qualification. In *DASIA 2015, The International Space System Engineering Conference*, Barcelona, June 2015. ASD-Eurospace.

[BS13a] A. Butterfield and D. Sanan. D03: Reference specification for a partitioning kernel and a separation kernel. Technical Report Version 8.1, Lero, 2013. ESTEC Contract No. 4000106016, comprising 3 documents: SRS, ADD, ICD.

[BS13b] A. Butterfield and D. Sanan. D04: Task 3: A formal verification process: approaches; feasibility; cost. Technical Report Version 4.0, Lero, 2013. ESTEC Contract No. 4000106016.

[BS13c] A. Butterfield and D. Sanan. D06: Evaluation of formal language, formal methods and modelling languages suitable for implementing an abstract specification. Technical Report Version 1.0, Lero, 2013. ESTEC Contract No. 4000106016.

[BS13d] A. Butterfield and D. Sanan. D07: Abstract Specification. Technical Report Version 4.0, Lero, 2013. ESTEC Contract No. 4000106016.

[BS13e] A. Butterfield and D. Sanan. D08: Assessment of the formal verification process. Technical Report Version 2.0, Lero, 2013. ESTEC Contract No. 4000106016.

[BSH13] A. Butterfield, D. Sanan, and M. Hinchey. Towards formal verification of a separation microkernel. In *DASIA 2013, The International Space System Engineering Conference*, Porto, May 2013. ASD-Eurospace.

[C+15] Loïc Correnson et al. *Frama-C User Manual*. CEA List, Software Safety Laboratory, Saclay, F-91191, release magensium-20151002 edition, 2015. `http://frama-c.com/download/frama-c-user-manual.pdf`.

[CDH+09] Ernie Cohen, Markus Dahlweid, Mark Hillebrand, Dirk Leinenbach, Michał Moskal, Thomas Santen, Wolfram Schulte, and Stephan Tobies. VCC: A practical system for verifying concurrent C. In Stefan Berghofer, Tobias Nipkow, Christian Urban, and Makarius Wenzel, editors, *Theorem Proving in Higher Order Logics, 22nd International Conference, TPHOLs 2009*, volume 5674 of *Lecture Notes in Computer Science*, pages 23–42, Berlin, August 2009. Springer-Verlag.

[CES86] Edmund M. Clarke Jr., E. Allen Emerson, and A. Prasad Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Transactions on Programming Languages and Systems*, 8(2):244–263, April 1986.

[Cri05] Common Criteria. Common criteria for information technology security evaluation, version 2.3. Technical Report ISO/IEC 15408:2005, Common Criteria, 2005.

[Dir07] Information Assurance Directorate. Protection profile for separation kernels in environments requiring high robustness. Technical report, U.S. Government, June 2007.

[GRABR14] 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.

[Hoa85] C. A. R. Hoare. *Communicating sequential processes*. computer science. Prentice-Hall International, Englewood Cliffs, N.J, 1985.

[NPW02] Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*, volume 2283 of *LNCS*. Springer, 2002.

[QS82] J. P. Queille and J. Sifakis. Specification and verification of concurrent systems in CESAR. In *Proceedings of the Fifth International Symposium in Programming*, volume 137 of *Lecture Notes in Computer Science*, pages 337–351, New York, 1982. Springer.

[Tre08] Jan Tretmans. Model based testing with labelled transition systems. In Robert M. Hierons, Jonathan P. Bowen, and Mark Harman, editors, *Formal Methods and Testing, An Outcome of the FORTEST Network, Revised Selected Papers*, volume 4949 of *Lecture Notes in Computer Science*, pages 1–38. Springer, 2008.