Controlled Owicki-Gries Concurrency: Reasoning about the Preemptible *eChronos* Embedded Operating System

June Andronick
NICTA and UNSW
june.andronick@nicta.com.au

Corey Lewis
NICTA
corey.lewis@nicta.com.au

Carroll Morgan
NICTA and UNSW
carroll.morgan@unsw.edu.au

We introduce a controlled concurrency framework, derived from the *Owicki-Gries* method, for describing a hardware interface in detail sufficient to support the modelling and verification of small, embedded operating systems (*OS*'s) whose run-time responsiveness is paramount. Such real-time systems run with interrupts mostly enabled, including during scheduling. That differs from many other successfully modelled and verified *OS*'s that typically reduce the complexity of concurrency by running on uniprocessor platforms and by switching interrupts off as much as possible.

Our framework builds on the traditional Owicki-Gries method, for its fine-grained concurrency is needed for high-performance system code. We adapt it to support explicit concurrency control, by providing a simple, faithful representation of the hardware interface that allows software to control the degree of interleaving between user code, *OS* code, interrupt handlers and a scheduler that controls context switching. We then apply this framework to model the interleaving behavior of the *eChronos* OS, a preemptible real-time *OS* for embedded micro-controllers. We discuss the accuracy and usability of our approach when instantiated to model the *eChronos* OS. Both our framework and the *eChronos* model are formalised in the Isabelle/HOL theorem prover, taking advantage of the high level of automation in modern reasoning tools.

1 Introduction

Formal verification is an inescapable requirement in cases where software/hardware failure would be catastrophic. Existing modelled and verified operating systems (e.g. [11, 17, 8, 3]) typically run on uniprocessor platforms. They are also *not preemptible*, i.e. they run with interrupts mostly disabled, at least during scheduling; thus their execution is mostly *sequential*.

Here, in contrast, we target *preemptible* (still uniprocessor) real-time *OS* code. Our motivating example is the *eChronos* OS [2], an open-source real-time *OS* that provides a library of *OS* services to applications, including synchronisation primitives (signals, semaphores, mutexes), context switching, and scheduling. Our approach, however, applies to any system where the *OS* code is preemptible, including scheduler code, and runs on uniprocessor hardware that supports nested interrupts. While being preemptible, the *OS* code is not *re-entrant*, which means that its execution can be interrupted at any moment by an interrupt handler servicing a hardware-device interrupt (unless that interrupt is masked off), but its execution is resumed after the interrupt has been handled. In order to allow faster response time, the *OS* is also *preemptive*, meaning that it can unilaterally take control from application tasks.

The *eChronos* OS is used in tightly constrained devices such as medical implants, running on embedded micro-controllers with no memory-protection support. It is small and comes in many variants. The variant we are targeting (which we will from now on simply refer to as the *eChronos* OS) runs on ARM uniprocessor hardware. [1] It makes use of ARM's *supervisor call* (*SVC*) mechanisms to run its scheduler,

[1] We specifically target an ARM Cortex-M4 platform, which, for the purposes of this paper, we will simply refer to as ARM.

Submitted to:
MARS 2015

where an SVC is a program-initiated interrupt, triggered by the execution of the SVC instruction, that results in the execution switching to an *OS*-provided SVC handler.

Earlier work has produced an initial formal specification of the *eChronos* API, but assumed that the execution of each API function was sequential, i.e. assumed that execution of interrupt handlers could not affect the API's functionality. Furthermore, it could not model the effect of context switching, which made proving refinement between this model and the (existing) implementation impossible.

That is what motivated the work presented here, where we focus on the interleaving behavior induced by unpredictable device interrupts and (predictable) context switching, but still provide a detailed, faithful model of the precise interleaved execution of user tasks, SVC handlers, and interrupt handlers, including nested ones. For wider usability we dissociate the general controlled-concurrency framework, and formal model of the API of the hardware mechanisms, from its specific instantiation to the model of the *eChronos* OS. We plan to then prove that this restricted but faithful model of the *eChronos* OS is refined by its implementation, and enrich the model with a complete specification of the API.

We follow the foundational Owicki-Gries (*OG*) concurrency method [14], where Hoare-style assertional reasoning is adapted to reason about a number of individually sequential processes that are executed collectively in parallel: the execution of the overall system is a non-deterministic interleaving of atomic statements each executed in the order determined by the process within which it occurs. Our choice of *OG* over more recent, derived concurrency styles, comes from the low-level of abstraction, needed for high-performance shared-variable system code. We model the *OS* system as the parallel composition of various user tasks (consisting of application code and calls to *OS* code), the interrupt handlers and the SVC handlers. On a uniprocessor platform, this allows much more interleaving that can happen in reality, where interleaving is controlled via hardware mechanisms such as context switching, enabling and disabling interrupts, etc. We adapt *OG* by adding an explicit control of interleaving, and we provide a formal hardware interface for operations manipulating allowed interleaving. We have formalised this framework in the Isabelle/HOL [12] theorem prover, building on an existing formalisation of *OG* [15].

In summary, we present the following contributions in this paper: (1) an adaptation of the *OG*-based concurrency model that controls interleaving; (2) a concise formal model of the API of the hardware mechanisms that control the interleaving induced by interrupts, SVC's and preemption; and (3) a model of the scheduling behavior of the *eChronos* OS. All of our work is formalised in Isabelle/HOL.

2 Explicit concurrency control in Owicki-Gries reasoning

The formalism we choose to represent interleaved execution in the small preemptible *OS* we aim to model, is based on the Owicki-Gries method, which we adapt to support explicit concurrency control.

The *OG* method extends Hoare logic for sequential programs [6] to concurrent programs that share data. An *OG* system comprises a number of *tasks* built from atomic statements. The concurrency between the tasks, i.e. an interleaving of atomic executions, is essentially uncontrolled except for the *await statement* with which a task can ensure its execution is suspended until a condition (of its choice) holds. Await statements are of the form *AWAIT C THEN P END* for some Boolean expression *C* in the system variables and some program fragment *P*: execution of *P* cannot occur unless *C* is (atomically) evaluated to true, in which case *P* is executed (also atomically) immediately afterward.

An *OG* proof generates verification conditions, *VC*'s, of two kinds: conventional post-then-pre conditions, and *interference-freedom VC*'s. The latter express that one task does not falsify, i.e. "interfere with", some conventional assertion in another task; it is essentially a non-compositional technique, however. Worse, those *VC*'s are quadratically numerous in the size of the program, which historically has

limited *OG*'s applicability to small systems.

Variants and extensions of *OG* include *rely-guarantee* [9] which addresses both compositionality and the number of *VC*'s. It also encourages a higher level of abstraction, which can sometimes impose execution-time inefficiency that might be intolerable in a high-performance application like the real-time preemptible *OS* we target here. (Compare while-loops, and invariant reasoning, with super-high performance low-level code that uses goto's in a less-structured way: sometimes –happily, not often– the latter is a necesary evil.) The same abstraction/performance tradeoff contra-indicates the use of more structured run-time mechanisms like monitors and critical regions [5, 7].

Targeting high-performance code is the reason why we chose the lower-level *OG* style. Since we aim to *verify* the *OS* systems we have modelled, we will eventually have to deal with the explosion of number of generated *VC*'s. We believe (and have initial evidence, discussed in §4) that for our application, and with our extension to control interleaving, mechanical verification is likely to overcome those difficulties.

Our extension follows from the observation that a uniprocessor *OS* is not truly concurrent: via interrupts and saved contexts it interleaves its tasks' executions in a way that simulates concurrency. Since our modelling *includes* that concurrency management, i.e. it includes the system's scheduler code, we must include the concurrency control in our program text. To allow an *OG* program to *control* its own interleaved concurrency, we associate a unique value with each task and we place each *OG*-atomic command in a task within an *AWAIT* condition requiring a global variable *AT* ("active task") be equal to the value associated with that task. Suppose for example we had a Task 2 whose atomic statements were *First*; *Second*; *Third*. It would become:

AWAIT AT=2 *THEN First END*; *AWAIT AT*=2 *THEN Second END*; *AWAIT AT*=2 *THEN Third END*;

Now if Task 2 were to give up control explicitly to, say some Task 1 similarly treated with *AWAIT AT*=1 decorations, it would simply include the (atomic) command *AWAIT AT*=2 *THEN AT* := 1 *END*; at the appropriate point. The basic *OG* mechanism then ensures that Task 1 continues execution from the point it last had control. In a more sophisticated system, Task 2 might transfer control instead to a "scheduler task", say Task 0, which would be a loop of the form (after pre-processing)

WHILE True DO
  AWAIT AT=0 *THEN t* := *"next runnable task" END*;
  AWAIT AT=0 *THEN AT* := *t END*
END

Our extended OG framework defines the pseudo-variable *AT* and uses it in the style above. The variable is *pseudo* in the sense that it does not represent a program variable, but rather some internal state managed by the hardware. We provide the function ***control*** that performs the automatic pre-processing step within Isabelle of inserting all the *AWAIT AT*=... code. It takes as input a task identifier T (2 in the example above) and the task's program text, and it adds *AWAIT*-statements with the guard *AT*=T to each atomic command of the program. The program above would become ***control*** 2 $\mathscr{P}$, where $\mathscr{P}$ is Task 2's program (here the three instructions shown). Our controlled interleaving will ensure that the majority of non-interference VC's will be trivial, [2] so that mechanised verification will allow automatic discharge.

3 Formalisation of the Hardware Interface

Our aim is to model uniprocessor, preemptible and preemptive software systems comprising user tasks, interrupt handlers, and ARM-provided SVC mechanisms. The model of §2 represents the interleaving

[2] A non-interference *VC* asserts that some assertion in task *X* is preserved by some statement from task *Y*. Since most assertions will be guarded by *AT*=*X*, and most statements will by *AT*=*Y*, many of these *VC*'s will have antecedent *False*.

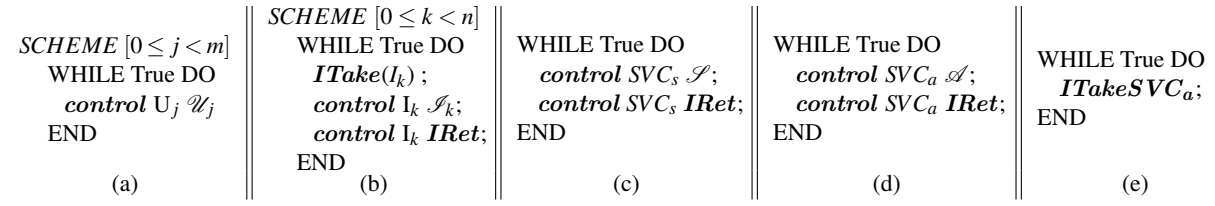

Figure 1: Model of OS-system with m user tasks and n interrupt handlers.

between tasks. We now formalise the hardware interface that allows the *OS* to control interleaving. The formalisation of the whole system is then presented in Fig. 1 and explained below.

3.1 Controlling interrupts

User tasks run application code that may call *OS* services; and with *user code* we refer to both.[3] Assuming we have user tasks U_1, ..., U_m, the code of task U_j will be noted $\mathscr{U}_j$. The user-task part of the system is represented by (a) in Fig. 1, where the notation $SCHEME\ [0 \leq j < m]\ c_j$, borrowed from [15], is the *parametric* representation of m parallel processes, i.e. $c_0 \mid\mid c_2 \mid\mid ... \mid\mid c_{m-1}$.

When an unmasked interrupt occurs, the running code is stopped, context saved, and interrupt-handler code starts instead. At the end of the execution of the interrupt handler, the hardware performs a return-from-interrupt instruction, restoring the saved context and switching back to the appropriate task. The code $\mathscr{I}_k$ for the interrupt task I_k might include potential *OS*-wrapper code. This interrupt part of the system is modelled by (b) in Fig. 1. Non-deterministic occurrence of interrupts is analogous to *OG*'s spontaneous (uncontrolled) task-switching, so the hardware mechanism that traps to the interrupt code (***ITake***) should *not* be guarded (and this is where interleaving happens). It should model the saving of context and updating of the *AT* variable to be I_k. In terms of context that needs to be saved, we only model here what is relevant to the task interleaving. The identity of the task being interrupted (i.e. the value of *AT*) needs to be saved (before being updated to I_k), to be able to return to it. On platforms like ARM, nested interrupts are allowed (i.e. interrupt handlers can be interrupted), so we save the whole stack of interrupted tasks. For this, we use a second pseudo-variable *ATstack*. In reality, interrupts can be *masked* using specific hardware functions, and may also obey some platform-dependent *interrupt policy*. Masked interrupts remain pending until they are unmasked. We use a third pseudo-variable *EIT* to represent the set of "enabled interrupt tasks", i.e. the hardware mask bits. This set can be manipulated using the following two functions, representing the hardware API to manipulate the interrupt mask.

$$Int\_Disable(X) \equiv EIT := EIT - X \quad (1) \qquad \text{and} \qquad Int\_Enable(X) \equiv EIT := EIT \cup X \quad (2)$$

These functions can be called inside *OS* code or interrupt code. Note that when an interrupt is re-enabled, the hardware checks whether that interrupt occurred while masked and is still pending, in which case it traps to the corresponding interrupt handler. The interesting point is that our modelling of interrupts, i.e. allowing interrupt code to non-deterministically run at any time, already represents this case, so adding the interrupt identifier to *EIT* is enough.

The interrupt policy defines the allowed nesting of interrupts. For instance, on ARM, interrupts can only be interrupted by a higher priority interrupt. In our model, we leave this platform-dependent policy generic, using a predicate $interrupt\_policy\ X\ Y$ which is true only if Y is allowed to interrupt X.

[3] A real-time *OS* typically runs as an *OS* library with no mode-switch, modelled here by *OS* code being inlined in the application code.

The full sequence of what happens when an interrupt occurs is that the hardware checks whether (i) the interrupt is unmasked (i.e. is in *EIT* in our model), (ii) the interrupt is not the same interrupt that is being processed (i.e. is not *AT* in our model), because if this happens the interrupt will remain pending as with any masked interrupt, (iii) the interrupt is allowed to interrupt the currently running task, according to the interrupt policy. If all these conditions are met, then the context is saved (now including the variable *EIT*, also saved on a stack *EITstack* for similar reasons as for *AT*) and control is switched to the interrupt handler. So in total the $\boldsymbol{ITake}$ function is defined as follows (where the operator $+$ pushes an element onto the stack):

$$\begin{array}{rcl} \boldsymbol{ITake}(X) & \equiv & AWAIT\ X \in EIT - AT - ATstack \ \wedge\ (interrupt\_policy\ AT\ X)\ THEN \\ & & \quad ATstack := AT + ATstack; \\ & & \quad EITstack := EIT + EITstack; \\ & & \quad AT := X; \\ & & END \end{array}$$

Note that all statements inside an *AWAIT* are executed atomically, where the atomicity here is ensured by the hardware (i.e. these multiple instructions represent a single atomic hardware-defined mechanism).

By virtue of our extended *OG* with *AWAIT*-guarded atomic statements, this enforces that the handler runs until further update of the *AT* variable, which only happens at the return from interrupt (or if another, unmasked, higher-priority interrupt occurs).

When returning from an interrupt *I*, the hardware will check whether there are any pending interrupts that would have happened during the handling of *I* but could not run because they were masked or because of the interrupt policy. Similarly to re-enabling interrupts, this case is already modelled by allowing non-deterministic interrupts at any time. So the return-from-interrupt function $\boldsymbol{IRet}$ only needs to model the context restore (updating *AT* and *EIT* with the top of *ATstack* and *EITstack* respectively, and popping *ATstack* and *EITstack*). As opposed to $\boldsymbol{ITake}$, $\boldsymbol{IRet}$ needs to be guarded (using our $\boldsymbol{control}$ mechanism) as is the rest of interrupt code, because it should only run at the end of the interrupt code.

$$\boldsymbol{IRet}(X) \ \equiv\ (AT + ATstack) := ATstack; \quad (EIT + ATstack) := EITstack;$$

3.2 Program control of preemption, and supervisor calls

So far we have modelled the hardware mechanisms that interleave user code and interrupt code. The ARM platform additionally provides mechanisms to do supervisor calls (*SVC*), both synchronous and asynchronous. SVC's are treated as program-initiated interrupts that are triggered by software calls to specific platform functions. Their effect is to switch the execution to specific *OS*-provided SVC-handler code. Asynchronous SVC is typically used to control *OS* code preemption to avoid re-entrance (because interrupt handlers would delay a call to the scheduler if the interrupted task is in *OS* code). Synchronous SVC is typically used for kernel calls on platforms supporting dual-mode. In the *eChronos* OS, where *OS* calls are just function calls, synchronous SVC is used for direct yielding from application. Here we present our model of the effect of these additional platform functions within the framework we introduced above. We assume code $\mathscr{S}$ (resp. $\mathscr{A}$) for the synchronous (resp. asynchronous) SVC handler. A *synchronous* SVC is triggered by a call (in user code) to a hardware API function *SVC_now*(). The effect of this function is to switch to the execution of $\mathscr{S}$. As with interrupts, the hardware will (atomically) save context onto the necessary stacks, and set *AT* to the identifier of the synchronous SVC task, noted SVC_s.

$$SVC\_now() \ \equiv\ \langle\ ATstack := AT + ATstack;\ EITstack := EIT + EITstack;\ AT := SVC_s\ \rangle \quad (3)$$

The $\langle\cdot\rangle$ notation models the atomic execution of the instructions, where the atomicity is here ensured by a hardware-enforced atomic mechanism. The SVC_s task is then modelled as running in parallel to user

code and interrupt code, represented by (c) in Fig. 1, with its code wrapped in *AWAIT*-statements using our ***control*** mechanism, and followed by a return from interrupt (restoring the stacks).

The code for the asynchronous SVC task is modelled the same way, (d) in Fig. 1, but the trigger is delayed. The hardware provides a function to request an asynchronous supervisor call, $SVC_a\_Request()$, whose effect is simply to set a bit SVC_aReq to $True$:

$$SVC_a\_Request() \equiv SVC_aReq := True \tag{4}$$

Then at some point in the future where this bit is set, and the SVC_a task is allowed to run (i.e. it is not masked, and it is allowed to interrupt the running task according to the interrupt policy), execution will switch to running $\mathscr{A}$. We model this by having a separate task, (e) in Fig. 1, running completely unguarded, constantly checking if an asynchronous supervisor call has been requested, and is allowed to run. If it is the case, it resets the SVC_aReq bit, saves the stacks, and switches to the SVC_a task:

$\boldsymbol{ITakeSVC_a} \equiv$
  $AWAIT\ SVC_aReq \wedge SVC_a \in EIT - AT - ATstack \wedge (interrupt\_policy\ AT\ SVC_a)\ THEN$
    $SVC_aReq := False;$
    $ATstack := AT + ATstack;$
    $EITstack := EIT + EITstack;$
    $AT := SVC_a;$
  END

The introduction of these software-triggered interrupts requires modifying our modelling of return from interrupt. Recall that in reality the hardware checks for pending interrupts, but in our model we don't need to model this, since we allow interrupt handlers to run at any time. However, in the case of the software-triggered SVC_a interrupt, we need explicitly to model that, on return from interrupt I, the hardware checks whether SVC_aReq is set, and whether SVC_a is allowed to run (it may have been manually removed from the EIT set). To know if SVC_a is allowed to run, we need to inspect the heads of $ATstack$ and $EITstack$ as these are the context of the task that was interrupted by I. The ***IRet*** function therefore becomes:

$\boldsymbol{IRet}(X) \equiv$
  $IF\ SVC_aReq \wedge SVC_a \in (hd\ EITstack) - AT - ATstack \wedge (interrupt\_policy\ (hd\ ATstack)\ SVC_a)$
  $THEN\quad EIT := hd\ EITstack;$
  $\qquad\quad AT := SVC_a;$
  $ELSE\quad (AT + ATstack) := ATstack;$
  $\qquad\quad (EIT + ATstack) := EITstack;$

Our formalisation of the hardware interface is given by the functions (1)-(4), available to the *OS* to control interleaving. Additionally, the functions ***ITake***, ***IRet*** and $\boldsymbol{ITakeSVC_a}$, together with our ***control*** pre-processing, as presented in Fig. 1, form our formal concurrency framework, to be instantiated to a specific *OS* by defining $\mathscr{U}$, $\mathscr{I}$, $\mathscr{A}$, and $\mathscr{S}$.

4 Discussing an Instantiation to a Model of the *eChronos* OS

For wider usability, we have so far presented a general controlled concurrency framework and formal model of an API of hardware mechanisms. We have instantiated this general framework to define a model of the *eChronos OS* scheduling behavior, where tasks are allocated priorities by the user, and the scheduler is in charge of enforcing that tasks are scheduled according to their priorities, i.e. its main functional property is that *whenever application code is executing, then it is the highest priority task.* One of our longer-term goals is to allow formal verification of such correctness properties. Our other aim is to validate our model against the real implementation by formal means (formal proof of

refinement). Validating the hardware abstractions still requires informal arguments, though. Here we discuss the modelling, its accuracy and its usability.

Given the framework described in the previous section, we need to instantiate $\mathscr{U}$, $\mathscr{I}$, $\mathscr{S}$, and $\mathscr{A}$. The instantiation is given in Appendix A, and the whole Isabelle/HOL model in Appendix B.

In creating this model there were several issues that we had to consider to convince ourselves, and more importantly the *eChronos* OS developers, that our model represents reality. The first was that the way in which we constrain the *OG* interleaving is accurate. In particular, we had to investigate when interrupts can occur and what the hardware does during an interrupt entry and return. We also had to ensure that anything we modelled as being atomic actually is atomic in reality. For the most part this involves the functionality provided by the hardware that we model, such as the $\boldsymbol{ITake}$ and $\boldsymbol{IRet}$ functions seen in §3.1. We believe that we have correctly captured the hardware interrupt behavior and atomicity, according to the ARM manual [1]. Additionally, we have been careful *not* to use *OG*'s $\langle\cdot\rangle$ atomic statement outside of the hardware interface. This way, the only remaining assumed atomicity is the one of single *OG* statements, which we will need to validate by refinement proofs when moving on to verification of our model.

Another important issue was the distinction between variables that are part of the *eChronos* OS and the pseudo-variables for hardware mechanisms. Care was needed to ensure that these hardware variables are only read and/or modified where allowed to, namely in the hardware API. Since we target devices with no memory protection, this requirement will have to be validated for the *eChronos* OS code and will remain an assumption for any user-provided applications (and could be checked using static analysis).

To justify our use of OG reasoning and to demonstrate that the mechanisation provided by Isabelle is sufficient to deal with scalability for system like the *eChronos* OS, we have begun initial verification of our model. As expected, at first there are a very large number of verification conditions: on the order of 10,000. However, by just defining a method that automatically removes any redundant conditions we can easily reduce this to under 500. The majority of these are then trivial enough to be automatically solved by standard Isabelle/HOL methods, with the final 10 conditions requiring human guidance. We believe that this number could be reduced even further by small improvements to the automation.

5 Related work and Conclusions

Frameworks for reasoning about shared variable programs have been around for more than 30 years. *OG* was the first one to be proposed [14]; much derived work has been done since, addressing specific requirements (compositionality [9], resource separation [13], etc). These frameworks have mainly been used to prove the correctness of concurrency algorithms or protocols. Here we target low-level high-performance *OS* code. Similarily, higher-level conceptual tools such as monitors [7] and conditional critical regions [5] decrease the proof burden, but impose a performance penalty, a trade-off usually worth making for clarity, except for minimal high-speed *OS*-kernel application where efficiency is crucial.

Formal verification of operating systems, kernels, and hypervisors has been the focus of important recent research (for which see [10] for an overview). Successfully verified systems generally either run on uniprocessor platforms with interrupts mostly disabled (e.g. [11, 17, 16]), or their verification does not take interrupts into account (e.g. [8]). In [3], a Hoare-logic-based framework is proposed to certify low-level system code involving interrupts and preemptive tasks, but the scheduler and context switching tasks are still executed with interrupts disabled, and interrupt handlers cannot be interrupted. In contrast, our work supports nested interrupts and a preemptible scheduler. A proof of correctness of the FreeRTOS scheduler is proposed in [4]; the proof does not include the context switch itself and

focuses on the scheduler policy (picking the next task). This is complementary to our work, where we leave the policy generic and assume it will pick the highest priority task.

To our knowledge, our extended OG framework with controlled concurrency is the first to support reasoning about low-level system code that is fully preemptible, including scheduler code, with support for nested interrupts. We have successfully instantiated it to formalise the scheduling behavior of the *eChronos* OS, a real-world, deployed, embedded *OS*. Our promising initial verification work indicates that we will be able to formally prove important functional properties involving complex concurrency reasoning about highly shared low-level variables.

References

[1] *ARM Infocenter*. Available at `http://infocenter.arm.com/`.

[2] *The eChronos OS*. Available at `http://echronos.systems`.

[3] Xinyu Feng, Zhong Shao, Yu Guo & Yuan Dong (2009): *Certifying low-level programs with hardware interrupts and preemptive threads*. Journal of Automated Reasoning 42(2-4), pp. 301–347, doi:10.1007/s10817-009-9118-9.

[4] Joao F Ferreira, Cristian Gherghina, Guanhua He, Shengchao Qin & Wei-Ngan Chin (2014): *Automated verification of the FreeRTOS scheduler in HIP/SLEEK*. International Journal on Software Tools for Technology Transfer 16(4), pp. 381–397, doi:10.1109/TASE.2012.45.

[5] Per Brinch Hansen (1972): *Structured Multiprogramming*. Communications of the ACM 15, pp. 574–578, doi:10.1145/361454.361473.

[6] C. A. R. Hoare (1969): *An Axiomatic Basis for Computer Programming*. Communications of the ACM 12, pp. 576–580, doi:10.1145/363235.363259.

[7] C. A. R. Hoare (1974): *Monitors: An Operating System Structuring Concept*. Communications of the ACM 17, pp. 549–557, doi:10.1145/355620.361161.

[8] Yanhong Huang, Yongxin Zhao, Longfei Zhu, Qin Li, Huibiao Zhu & Jianqi Shi (2011): *Modeling and verifying the code-level OSEK/VDX operating system with CSP*. In: Theoretical Aspects of Software Engineering (TASE), 2011 Fifth International Symposium on, IEEE, pp. 142–149, doi:10.1109/TASE.2011.11.

[9] C. B. Jones (1983): *Tentative steps towards a development method for interfering programs*. ACM Transactions on Programming Languages and Systems 5(4), pp. 596–619, doi:10.1145/69575.69577.

[10] Gerwin Klein (2009): *Operating System Verification — An Overview*. Sādhanā 34(1), pp. 27–69, doi:10.1007/s12046-009-0002-4.

[11] Gerwin Klein, June Andronick, Kevin Elphinstone, Toby Murray, Thomas Sewell, Rafal Kolanski & Gernot Heiser (2014): *Comprehensive Formal Verification of an OS Microkernel*. ACM Transactions on Computer Systems 32(1), pp. 2:1–2:70, doi:10.1145/2560537.

[12] Tobias Nipkow, Lawrence Paulson & Markus Wenzel (2002): *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*. 2283, doi:10.1007/3-540-45949-9.

[13] Peter W. OHearn (2007): *Resources, Concurrency, and Local Reasoning*. Theor. Comput. Sci. 375(1-3), pp. 271–307, doi:10.1016/j.tcs.2006.12.035.

[14] Susan Owicki & David Gries (1976): *An axiomatic proof technique for parallel programs*. Acta Informatica 6, pp. 319–340, doi:10.1007/BF00268134.

[15] Leonor Prensa Nieto (2002): *Verification of parallel programs with the Owicki-Gries and rely-guarantee methods in Isabelle/HOL*. Ph.D. thesis, Technische Universität München.

[16] Raymond J. Richards (2010): *Modeling and Security Analysis of a Commercial Real-Time Operating System Kernel*, pp. 301–322. Springer US, doi:10.1007/978-1-4419-1539-9_10.

[17] Jean Yang & Chris Hawblitzel (2010): *Safe to the last instruction: automated verification of a type-safe operating system*. Toronto, Ontario, Canada, pp. 99–110, doi:10.1145/1806596.1806610.

A Instantiation of our Controlled OG Framework to the *eChronos* OS

Instantiating the framework described in §3 to the *eChronos* OS requires to instantiate $\mathscr{U}$, $\mathscr{I}$, $\mathscr{S}$, and $\mathscr{A}$. The instantiation is as follows.

$$\begin{array}{lcl}
\mathscr{I} & \equiv & E := \mathit{change_events}; \\
 & & \mathit{SVC_a_Request}(); \\
\mathscr{U}_j & \equiv & \mathit{syscall_block} \equiv \mathit{SVC_aDisable}(); \\
 & & R := R(j := \mathit{False}); \\
 & & \mathit{SVC_now}(); \\
 & & \mathit{SVC_aEnable}(); \\
 & & \mathit{WHILE} \neg \mathit{SVC_aReq\ DO\ \ SKIP\ \ END}; \\
\mathscr{S} & \equiv & \mathit{schedule}; \\
 & & \mathit{context_switch\ True}; \\
\mathscr{A} & \equiv & \mathit{schedule}; \\
 & & \mathit{context_switch\ False};
\end{array}$$

Since we are focusing on the scheduling behavior, we only model the parts that may influence the scheduling decisions, i.e. deciding which task should be the next to run. These decisions depend on (i) which are the runnable tasks, and (ii) the set of events signaled by interrupt handlers, which may influence which tasks are runnable. We use the variable R for the mapping from task identifier to a Boolean value indicating whether the task is runnable, and the variable E for the set of events.

The interrupt code $\mathscr{I}$ is mainly a user-provided interrupt handler, which is only allowed to call one specific *OS* function to change the set of events. Since this might change which tasks are runnable the scheduler needs to run to update the set of runnable tasks and potentially switch to a higher priority task. To avoid re-entrant *OS* code, the interrupt handler only flags the need for the scheduler to run (by requesting an asynchronous system call). This request must be handled before application code is run again. The function *change_events* represents a non-deterministic update. The rest of the interrupt handler's functionality is not represented, as it should not be relevant to the scheduling behavior.

$\mathscr{S}$ and $\mathscr{A}$ are almost identical and represent the scheduler code. The main job of the scheduler is to pick a new task to run, by first updating the runnable mapping R taking into account the set of unprocessed events E, and then picking the task to run according to the scheduling policy in place. Once the task is chosen, a context switch is performed, storing the old task and placing the new task on the stack. The full details of how *schedule* and *context_switch* are modelled can be found in §B.5.

Finally, the majority of the *eChronos* OS code is in $\mathscr{U}$, which represents application code (kept generic here) and calls to any of the *OS* API functions. We model application code only as potentially making an *OS* call, and we only model a single call that is representative of how the variables that we are interested in can be modified. The block syscall modifies R so that task U_j is not runnable and then yields by invoking $\mathrm{SVC_s}$ via *SVC_now*. To ensure that it is not re-entrant, the *OS* call is wrapped between disable and enable functions for the $\mathrm{SVC_a}$ interrupt and is followed by a loop waiting for $\mathit{SVC_aReq}$ to be set to *False*. As $\mathrm{SVC_a}$ is the only routine that sets $\mathit{SVC_aReq}$ to *False*, this ensures that, if required, $\mathrm{SVC_a}$ executes before control is returned to the user application. The functions $\mathit{SVC_aDisable}()$ and $\mathit{SVC_aEnable}()$ are defined as follows.

$$\begin{array}{lcl}
\mathit{SVC_aDisable}() & \equiv & \mathit{EIT} := \mathit{EIT} - \mathit{SVC_a} \\
\mathit{SVC_aEnable}() & \equiv & \mathit{EIT} := \mathit{EIT} \cup \mathit{SVC_a}
\end{array}$$

B Formal model of the eChronos *OS* scheduling behaviour in Isabelle

We present here a model of the ARM Cortex-M4 version of the eChronos OS scheduling behaviour, formalised in Isabelle/HOL. It is based on Leonor Prensa's formalisation of Owicki-Gries in Isabelle/HOL.

B.1 State

A routine is just a natural number; we add routines for both the SVC_s handler and the SVC_a handler, user routines have numbers from *2* to *nbUsers*+*2*(excluded) and interrupt routines have numbers from *nbUsers*+*2* to *nbUsers*+*nbInts*+*2* (excluded). The first user to run is arbitrarily chosen to be the first one.

type-synonym *routine* = *nat*

consts *nbUsers* :: *nat*
consts *nbInts* :: *nat*

abbreviation *nbRoutines* ≡ *nbUsers*+*nbInts*

abbreviation $SVC_s \equiv 0$
abbreviation $SVC_a \equiv 1$

definition *user0* ≡ *2*
definition $U \equiv \{user0..<user0 + nbUsers\}$
definition $I \equiv \{user0 + nbUsers..<user0 + nbUsers + nbInts\}$
definition $I' \equiv I \cup \{SVC_a\}$

A state is composed of all the hardware variables plus the program variables that the targeted invariant or property relies on.

record *'a state* =

EIT :: *routine set*	— the set of enabled interrupt tasks
SVC_aReq :: *bool*	— the SVC_a requested bit
AT :: *routine*	— the active routine
ATStack :: *routine list*	— the stack of suspended routines
curUser :: *routine*	— current user task
contexts :: *routine* ⇒ (*bool* × *routine list*) *option*	— stored contexts
R :: *routine* ⇒ *bool option*	— Runnable threads
E :: *nat set*	— Events set (current)
E-tmp :: *nat set*	— Temporary events set
nextT :: *routine option*	— the next Task

B.2 Controlled Owicki-Gries reasoning

The model of parallel composition allows more interleaving than the real execution, where only enabled routines can run. To model this we extend the OG formalisation with our controlled concurrency mechanism; we use the *AT* variable and wrap every instruction of routine *r* in an *AWAIT* $\{\!|AT = r|\!\}$ statement. The function *add-await-bare-com* performs this process. It recursively traverses the command tree, using the given property to construct the *AWAIT* statement which is added as required. The full definition of *add-await-bare-com* is not shown here.

definition *control*
where
control r c ≡ *add-await-bare-com* {|*AT* = *r*|} *c*

B.3 Generic scheduling policy, handling of events and interrupt policy

The scheduling policy (picking the next thread, given the list of runnable threads) is left unspecified here; as well as the updating of this runnable list, given a list of events. The interrupt policy (which interrupts are allowed to run, given the currently running routine) is also left unspecified.

consts *sched-policy* :: (*routine* ⇒ *bool option*) ⇒ *routine option*
consts *handle-events* :: *nat set* ⇒ (*routine* ⇒ *bool option*) ⇒ *routine* ⇒ *bool option*
consts *interrupt-policy* :: *routine* ⇒ *routine set*

B.4 A model of hardware interface

The following two functions are used to enable and disable the SVC_a interrupt. They do this by either adding or removing SVC_a from the EIT set.

definition
SVC$_a$*Enable*
where
SVC$_a$*Enable* ≡ *EIT* := *EIT* ∪ {*SVC*$_a$}

definition
SVC$_a$*Disable*
where
SVC$_a$*Disable* ≡ *EIT* := *EIT* − {*SVC*$_a$}

ITake *i* models the hardware mechanism that traps to the handler for interrupt *i*. First it checks whether the interrupt is enabled, is not already being handled and is a higher priority than the current routine. When these conditions are satisfied then the context[4] of the previous task is saved on a stack and AT is set to *i*.

definition
ITake
where
ITake i ≡
AWAIT i ∈ *EIT* − {*AT*} − *set ATStack* ∧ *i* ∈ *interrupt-policy AT*
THEN
⟨*ATStack* := *AT* # *ATStack*,, *AT* := *i*⟩
END

Similarly to above, ***SVC*$_a$ *Take*** models the hardware mechanism that traps to the SVC_a handler. It is almost exactly the same as ***ITake*** *i*, but because we can observe when SVC_a is requested we now also require that *SVC*$_a$*Req* is True before it can begin executing. It also sets *SVC*$_a$*Req* to False while setting AT to SVC_a.

definition

[4] We only model the part of the context relevant to controlling the interleaving. Here that is just the previous value of AT, which can be thought of as corresponding to the program counter. Note that this is in contrast to the model from §3.1, which also stores the value of EIT. This is because ARM does not save the mask status when an interrupt occurs, and it is up to the interrupt handlers to ensure that the interrupt mask is preserved.

SVC_aTake
where
$SVC_aTake \equiv$
$AWAIT\ SVC_aReq \wedge SVC_a \in EIT - \{AT\} - set\ ATStack \wedge SVC_a \in interrupt\text{-}policy\ AT$
$THEN$
$\langle ATStack := AT\ \#\ ATStack,,$
$AT := SVC_a,, SVC_aReq := False\rangle$
END

IRet models the hardware mechanism used to return from an interrupt. The main action it performs is to restore the context of the interrupted routine. It does this by setting AT to the head $ATStack$, which is then removed from the stack. However, if there is a pending interrupt that is now allowed to run then ***IRet*** will transfer control directly to this interrupt instead of restoring the stored context. Due to the construction of our model we only need to ensure that this happens for SVC_a, as it is the only interrupt that we can observe has occured.

definition
$IRet$
where
$IRet \equiv$
$\langle IF\ SVC_aReq \wedge SVC_a \in EIT - set\ ATStack \wedge SVC_a \in interrupt\text{-}policy\ (hd\ ATStack)$
$THEN\ AT := SVC_a,, SVC_aReq := False$
$ELSE\ AT := hd\ ATStack,, ATStack := tl\ ATStack$
$FI\rangle$

When SVC_now is called it triggers an SVC_s interrupt to occur, which is then immediately handled. The effect of this function is similar to that of ***ITake***, the active task is saved on the stack and then AT is set to SVC_s. We implicitly assume that SVC_s is enabled when SVC_now is called, as if this was not true in reality then the hardware would trigger an abort exception.

definition
$SVC\text{-}now$
where
$SVC\text{-}now \equiv \langle ATStack := AT\ \#\ ATStack,, AT := SVC_s\rangle$

$SVC_aRequest$ is used to request that SVC_s occurs as soon as it is next possible.

definition
$SVC_aRequest$
where
$SVC_aRequest \equiv SVC_aReq := True$

B.5 Model of the eChronos OS

The eChronos OS uses SVC_s and SVC_a interrupt handlers to implement scheduling. The scheduler function chooses a new task to run by first updating the runnable mapping R before using whichever scheduler policy is in place to pick a task from among the runnable ones. To update the runnable mapping, the function $handle\_events$ is used, with this function being left non-deterministic. After the execution of this function, the variable E needs to be cleared to indicate that the events have been processed. However, the scheduler may itself be interrupted. If an interrupt occurs between the execution of $handle\_events$ and the reset of E, the interrupt handler might have modified E with new events to be processed (and flagged a request for the scheduler to run). On return from interrupt, because the scheduler is itself an interrupt

and is not re-entrant, its execution resumes, and so E should not be cleared. Instead we save its value before running *handle_events*, and only remove those saved events that have indeed been processed. When the scheduler will return, the hardware will check if there are still any pending requests for the scheduler to run, and re-run it if required.

definition
schedule
where
schedule $\equiv$
nextT := *None*;;
WHILE nextT = *None*
DO
E-tmp := *E*;;
R := *handle-events E-tmp R*;;
E := *E* − *E-tmp*;;
nextT := *sched-policy*(*R*)
OD

Once the schedule functions has executed, the *context_switch* function is called. This function, as the name suggests, saves the context of the old user task that was previously on the hardware stack, and then replaces it with the context of the task chosen by the scheduler. To do this the function first stores whether the previous user task had SVC_a enabled,[5] along with the current value of *ATStack*. It then loads the stack that existed when the new task was last executing. Lastly, SVC_a is enabled or disabled as required by the new task.

definition
context-switch
where
context-switch preempt-enabled $\equiv$
contexts := *contexts* (*curUser* $\mapsto$ (*preempt-enabled*, *ATStack*));;
curUser := *the nextT*;;
ATStack := *snd* (*the* (*contexts* (*curUser*)));;
IF fst (*the* (*contexts* (*curUser*)))
THEN SVC_a*Enable*
ELSE SVC_a*Disable*
FI

Finally, we combine everything to construct the full eChronos OS model. First, the state is initialised with the correct starting values. Following this, the various routines are run in parallel, with concurrency controlled as required through the use of *control*.

definition
eChronos-OS-model
where
eChronos-OS-model change-runnables change-events $\equiv$
(*EIT* := *I′*,,
SVC_a*Req* := *False*,,
AT := *user0*,,
ATStack := [],,
curUser := *user0*,,

[5] This is identified by the boolean passed to context_switch. If *context_switch* is being called by SVC_a then clearly SVC_a was previously enabled, while by design we know that SVC_s is only called when SVC_a is disabled.

```
contexts := (λn. if n∈U then Some (True, [n]) else None),,
R := (λn. if n∈U then Some True else None),,
E := {},,
E-tmp := {},,
nextT := None,,
(COBEGIN
 (∗ SVC_a-take ∗)
 WHILE True
 DO
  SVC_aTake
 OD

 ||

 (∗ SVC_a ∗)
 WHILE True
 DO
  (control SVC_a (
  schedule;;
  context-switch True;;
  IRet))
 OD

 ||

 (∗ SVC_s ∗)
 WHILE True
 DO
  (control SVC_s (
  schedule;;
  context-switch False;;
  IRet))
 OD

 ||

 SCHEME [0 ≤ i < nbRoutines]
 IF (i∈I) THEN

 (∗ Interrupts ∗)
 WHILE True
 DO
  ITake i;;

  (control i (
  E := change-events;;
  SVC_aRequest;;

  IRet))
 OD

 ELSE
```

$$
\begin{array}{l}
(* \mathit{Users} *) \\
\mathit{WHILE}\ \mathit{True} \\
\mathit{DO} \\
\quad (\mathit{control}\ i\ (\\
\quad \mathit{SVC}_a\mathit{Disable};; \\
\quad R := R\ (i \mapsto \mathit{False});; \\
\quad \mathit{SVC\text{-}now};; \\
\quad \mathit{SVC}_a\mathit{Enable};; \\
\quad \mathit{WHILE}\ \neg \mathit{SVC}_a\mathit{Req} \\
\quad \mathit{DO} \\
\quad\quad \mathit{SKIP} \\
\quad \mathit{OD})) \\
\mathit{OD} \\
\mathit{FI} \\
\mathit{COEND}))
\end{array}
$$

Acknowledgements NICTA is funded by the Australian Government through the Department of Communications and the Australian Research Council through the ICT Centre-of-Excellence Program.