Proof of OS scheduling behavior in the presence of interrupt-induced concurrency

June Andronick[1,2], Corey Lewis[1], Daniel Matichuk[1,2],
Carroll Morgan[1,2], and Christine Rizkallah[1,2]

[1] Data61, CSIRO (formerly NICTA) `firstname.lastname@data61.csiro.au`
[2] UNSW `firstname.lastname@unsw.edu.au`

Abstract. We present a simple yet scalable framework for formal reasoning and machine-assisted proof of interrupt-driven concurrency in operating-system code, and use it to prove the principal scheduling property of the embedded, real-time *eChronos OS*: that the running task is always the highest-priority runnable task. The key differentiator of this verification is that the *OS* code itself runs with interrupts *on*, even within the scheduler, to minimise latency. Our reasoning includes context switching, interleaving with interrupt handlers and nested interrupts; and it is formalised in Isabelle/HOL, building on the Owicki-Gries method for fine-grained concurrency. We add support for explicit concurrency control and the composition of multiple independently-proven invariants. Finally, we discuss how scalability issues are addressed with proof engineering techniques, in order to handle thousands of proof obligations.

1 Introduction

We address the problem of providing strong machine-checked guarantees for (uniprocessor) operating-system code with a high-degree of interrupt-driven concurrency, but without hardware-enforced memory protection. Our contribution is a technique to reason feasibly about preemptive real-time kernels; we demonstrate its effectiveness on the commercially-used embedded *eChronos OS* [3].

Rather than inventing our own, new concurrency formalism from scratch, we chose to "go for simplicity". The Owicki-Gries method [19], more than 40 years old, was invented when mechanised theorem proving was scarce, when the principal concern was compact, elegant formalisms applied to small intricate problems.[3] Proving an elegant property of a small, intricate operating system thus seemed to be an ideal experiment; an added attraction was that the system is in real-world use. We further motivate the choice of Owicki-Gries in §4.1.

There were, however, two immediate concerns: even a small *operating system kernel* is nowhere near small enough to be reasoned about by hand, as the *OG* (Owicki-Gries) pioneers typically did [8]; and the *OG* concurrency model did not at first seem right for reasoning about the coarse-grained concurrency of switching between tasks. In *OG*, atomic actions are arbitrarily interleaved, and *OG*'s

[3] Broadly speaking, this was the "Hoare/Dijkstra/Gries School" of Formal Methods.

"await statements" (§2 below) are not designed for reasoning about interrupt-driven scheduling *including* the scheduler and context-switching code itself.

The former concern would, we hoped, be taken care of by the increased power and sophistication of theorem provers in the decades since *OG* was introduced: we use Isabelle/HOL [18]. The latter concern is handled by our novel style of *OG* reasoning, presented in a previous paper [4], that adapts *OG* to allow reasoning about interrupt-induced and scheduler-controlled concurrency. Although conceptually simple, this style introduces significant extra text that we hide through modern techniques: we call it "await painting". Await-painting introduces an active-task variable that tracks which task is allowed to execute, and wraps every atomic statement with an *AWAIT* using that variable to restrict the allowed interleavings. It is what allows the program code itself to control the interleaving between tasks, something not normally done in *OG*.

The top-level theorem we prove is a scheduling property, not directly expressible in *OG* without the await-painting step: that the currently executing task is the highest-priority runnable task. We prove it on a model of the interleaving between the *OS* and the (possibly nested) interrupt handlers. In future work, we aim to prove that this model is a correct abstraction of the existing *eChronos OS* implementation. Our proof assumes that all application-provided code is well behaved; that is, it does not change any *OS* variables, and application-provided interrupt handlers only call a specific API function. Although the *eChronos OS* does not explicitly export any functions modifying *OS* variables, the *OS* is not able to enforce these constraints as the system runs on hardware with no memory protection. These assumptions can, however, be statically checked.

Our specific contributions are the following ones. We extend the model presented previously [4] (Contribution 2), while the proofs themselves are new (Contributions 1, 3, 4, 5). Most of our model and proof framework is generic and should apply to systems that support interruptible OS-es, preemptible applications and nested interrupts. All our proofs and model are available online [1].

1. We provide a proof framework, using a formalisation of *OG* in Isabelle/HOL [20], to reason about interrupt-driven and scheduler-controlled concurrency. Our framework is driven by the aim to handle complex parallel composition which requires that invariants can be proved compositionally. (§4.1)
2. We give an updated model of our interleaving framework and instantiation to the *eChronos OS*. It extends the one presented previously [4] by including system calls that can influence the scheduling decisions, introduces non-determinism to properly represent under-specified operations, and properly separates generic interleaving and *eChronos* instantiation. (§3)
3. We show that proving the scheduling property for the *eChronos* system is within the capabilities of modern theorem provers, at least for an application of this size. This includes handling the *OG*-characteristic of quadratically many "interference-freedom" verification conditions. (§4.2)
4. We develop a number of proof engineering techniques to address some observed problems that occur in a proof of this scale. (§4.3)
5. We contribute to the real-world utility of an existing preemptive kernel that is in widespread use, in particular in medical devices.

2 Background and big picture

The goal of our work is to provide a verification framework for *OS* code involving interrupt-induced concurrency, in particular real-time embedded *OS*-es.

A real-time *OS* (RTOS), like the *eChronos OS*, is typically used in tightly constrained embedded devices, running on micro-controllers with limited memory and no memory-protection support. The role of the *OS* is closer to that of a library than of a fully-fledged operating environment, allowing the application running on top to be organised in multiple independent tasks and providing a set of API functions that the application tasks can call to synchronise (signals, semaphores, mutexes). The *OS* also provides the underlying mechanism for switching from one task to another, and is responsible for sharing the available time between tasks, by scheduling them according to some given *OS*-specific policy. For instance, tasks can cooperatively yield control to each other (*cooperative scheduling*); or tasks can be scheduled according to their assigned *priority*, and their execution must then be *preempted* if a higher priority is made available (*preemptive scheduling*). The system typically also reacts to external events via interrupts. An *interrupt handler* needs to be defined for each interrupt by the application. When an interrupt occurs, the hardware ensures that the corresponding interrupt handler is executed (unless the interrupt is disabled/masked).

The job of the scheduler is to ensure that at any given point the running task is the correct one, as defined by the scheduling policy of the system.

For instance, in a priority-based preemptive system, when a task in unblocked (e.g. by an interrupt handler sending the signal it was waiting for) a context switch should occur if this task is at a higher priority than the currently running one. This defines the *correctness of the scheduling behavior* and is the target of our proof about the *eChronos OS*.

To reason about such an RTOS, and prove such a scheduling property, we present a *verification framework supporting the concurrency reasoning required by preemption and interrupt handling* (on uniprocessor hardware).

In previous work [4] we provided a model of interleaving that faithfully represents the interaction between application code, OS code, interrupt handler and scheduler, in such an RTOS. Roughly, the system is modelled as a parallel composition $A_1||...||A_n||Sched||H_1||...||H_m$, where the code for each application A_i is parameterised (including calls to *OS* API functions), as well as the code for each interrupt handler H_j, and the code for the scheduler. The key feature of the framework is that the interleaving in the parallel composition is *controlled* using a small formalised API of the hardware mechanisms for taking interrupts, returning from interrupts, masking interrupts, etc. We have formalised our logic in Isabelle/HOL, based on [20].

In this paper we present (a) a logic to prove invariants about such parallel composition, with support for handling complex proofs and (b) instantiation of the model to the *eChronos OS* and proof of its scheduling behavior.
Namely, the property we prove is

$$\Vdash_b \{\!|\textit{scheduler-invariant}|\!\}\ \{\!|\textit{True}|\!\}\ \textit{eChronos-sys}\ \{\!|\textit{False}|\!\} \qquad (1)$$

where $\Vdash_b$ is the derivability of a "bare" program (i.e. with no annotations), and is defined in terms of $\|-_i$ at the end of §4.1. The notation $\|-_i\ I\ p\ c\ q$ means that if the (annotated) parallel program c starts in a state satisfying the precondition p, then the invariant I holds at all reachable execution steps of c, and the postcondition q holds if c terminates. The definition, explained in detail in §4.1, adds an invariant to the original Owicki-Gries statement $\|-\ p\ c\ q$, which in turn is an extension of traditional Hoare-logic statement $\vdash\ p\ c\ q$. Owicki-Gries extends the sequential programs of Hoare-logic with two constructs: the parallel composition $c_1||c_2$ and the *AWAIT*-statement $AWAIT\ b\ DO\ c\ OD$. The execution of $c_1||c_2$ is the execution of the current instruction of *either* c_1 *or* c_2. The statement $AWAIT\ b\ DO\ c\ OD$ can only execute if condition b is satisfied, in which case c is executed *atomically* (meaning that b is still true as c begins, and reasoning within c is purely sequential).

eChronos_sys is our model of an *eChronos* system. The *eChronos OS*[4] provides a priority-based preemptive scheduler with static priorities. It comprises about 500 lines of C code and runs on ARM uniprocessor hardware.[5] Our model is an instantiation of the generic model of interleaving [4], with definitions for the scheduler, and for the API system calls (the ones that may influence the scheduling decisions) which are called from application or handler code. This model is given in §3.

Coming back to the property (1), it says that the *eChronos* system, starting in *any* initial state and never terminating, will satisfy the *scheduler_invariant* at every point of execution. The precondition *True* is always trivially satisfied and the postcondition *False* is valid because the system is an infinite loop of execution. The invariant property for the *eChronos OS* states that *the running task is always the highest priority runnable task*. We describe its formal definition and proof in §4.2.

Owicki-Gries reasoning introduces quadratically many proof obligations due to parallelism: indeed our proof for the *eChronos* scheduling behavior initially generates thousands. However, using a combination of (a) compositional proofs for proving invariants; (b) controlled interleaving to eliminate unfeasible executions; and (c) proof engineering techniques to automate discharging a large number of conditions, we show the feasibility of this approach for a preemptive and interruptible real-time OS running on a uniprocessor §4.3.

3 The model

In recent work [4] we presented a model of interleaving between application code, interrupt handler code, and scheduler code, for an ARM platform that supports

[4] The *eChronos OS* [3] comes in many variants, varying in the hardware they run on, the scheduling policy they enforce and the synchronisation primitives they offer. In this paper we simply refer to the *eChronos OS* for the specific variant that we are targeting, called *Kochab*, which supports the features that create interesting reasoning challenges (preemption, nested-interrupts, etc).

[5] We specifically target an ARM Cortex-M4 platform, simply referred to as ARM here.

both direct and delayed calls to the scheduler. The model was designed to be generic and we then instantiated it to the *eChronos OS*.

Here we present this model, with several improvements. We explicitly separate the generic portion to clarify how one could use the framework to formalise a different system. We extended the formalisation of the *eChronos OS* to include system calls that can influence the scheduling decisions. Finally, we introduce non-determinism to properly represent under-specified operations.

3.1 A generic model of interrupt-driven interleaving

In the generic part of the model we focus both on formalising the hardware mechanisms that control interleaving and on faithfully representing the concurrency induced by interrupts. The system is modelled as the parallel composition $svc_aTake||svc_a||svc_s||H_1||...||H_m||A_1||...||A_n$, where the scheduler (*Sched* in §2) is taking into account here both direct/synchronous calls to the scheduler (svc_s) and delayed/asynchronous ones (svc_a). ARM provides a direct (synchronous) *supervisor call* (*SVC*) mechanism that can be thought of as a program-initiated interrupt. It is triggered by the execution of the SVC instruction (SVC_now), which results in the execution switching to an SVC handler (svc_s). ARM also provides a delayed (asynchronous) supervisor call, also behaving like an interrupt, with instructions allowing programs to enable and disable it. It is triggered by raising a flag (svc_aReq), whose status is constantly checked by the hardware (modelled by svc_aTake). If the flag is raised and the asynchronous SVC is enabled, the execution will switch to a specific handler (svc_a). In the case of the *eChronos OS*, both SVC handlers will execute the scheduler.

The formal model of interleaving is presented in Figure 1.[6] The code for the application initialisation, application tasks, interrupt handlers and SVC handlers are parameters as they are system-specific.

The code for each part of the parallel composition is in fact wrapped in an infinite *WHILE* loop, reflecting the reactive nature of the system. Moreover, to faithfully represent the controlled interleaving allowed by the hardware we *await-paint* most of the code. This means that we introduce an active-task variable, AT, and associate each task, including the interrupt handlers, with a unique identifier. Every atomic statement c in Task t is then converted into a statement *AWAIT AT=t THEN c END*. As described in §2 this prevents the execution of c until the await-condition holds. Only when the command *AT:= t* is performed will Task t be able to execute. In particular, this means that no other Task t' with $t' \neq t$ will be able to interfere. In the model this await-painting is performed by the *control* function, which recursively wraps every command with an *AWAIT*.

[6] The model is written in a simple formalised imperative language with parallel composition and await statements, which has the following syntax:

$$c \equiv x := v \mid c;;\ c \mid \mathit{IF}\ b\ \ \mathit{THEN}\ c\ \ \mathit{ELSE}\ c\ \mathit{FI} \mid \mathit{WHILE}\ b\ \mathit{DO}\ c\ \mathit{OD} \mid \\ \mathit{AWAIT}\ b\ \ \mathit{THEN}\ c\ \mathit{END} \mid (\mathit{COBEGIN}\ c \parallel c\ \mathit{COEND})$$

The *SCHEME* constructor models a parametric number of parallel programs, as seen in [20]. Here this is the number of handlers plus the number of application tasks.

```
definition interleaving app-init $svc_a$-code $svc_s$-code handler-code app-code ≡
  hardware-init;;
  app-init;;
 (COBEGIN
    WHILE True DO $svc_a$Take OD
  ∥ WHILE True DO control $svc_a$ $svc_a$-code OD
  ∥ WHILE True DO control $svc_s$ $svc_s$-code OD
  ∥ SCHEME [user0 ≤ i < user0 + nbRoutines]
    IF i ∈ I THEN
      WHILE True DO ITake i;; control i (handler-code i) OD
    ELSE
      WHILE True DO control i (app-code i) OD
    FI
 COEND)
```

Fig. 1. Definition of generic interrupt-driven interleaving in Isabelle/HOL

To be precise, we await-paint all of the code except for where concurrency can actually occur: during the background hardware routine svc_aTake, and when an interrupt is taken, $ITake$. These represent our model of the hardware mechanisms that control the interleaving and context switching. We define them as below, with an $AWAIT$ with the condition that the interrupt is allowed to be taken. We have previously described these functions in detail [4], but abstractly they save the AT variable on a stack (the notation $x\#xs$ adds x to the list xs) and switch to the interrupt or SVC handler. While these are the only places where concurrency is not controlled, they are still guarded by the condition that the interrupt is enabled (is in the set EIT of enabled interrupts), is not already running (or itself interrupted), and is allowed to interrupt the active task.

```
can-interrupt i ≡ i ∈ EIT − ATStack ∧ i ∈ interrupt-policy AT

ITake i ≡ AWAIT can-interrupt i
THEN ATStack := AT # ATStack;; AT := i END

$svc_a$Take ≡ AWAIT $svc_a$Req ∧ can-interrupt $svc_a$
THEN ATStack := AT # ATStack;; AT := $svc_a$;; $svc_a$Req := False END
```

One of the central features of the $eChronos$ OS is that while OS code is interruptible, it is not preemptible. In practice, this means that although standard interrupts are handled immediately, the call to the scheduler via the $\mathrm{SVC_a}$ interrupt is delayed until the OS code is completed. To achieve this, $\mathrm{SVC_a}$ is temporarily removed from EIT, ensuring that svc_aTake cannot execute.

We also provide a model of the SVC_now and $IRet$ hardware instructions that are called by OS functions.

```
SVC-now ≡ ATStack := AT # ATStack;; AT := $svc_s$

IRet ≡ IF $svc_a$Req ∧ can-interrupt′ $svc_a$
THEN AT := $svc_a$;; $svc_a$Req := False
ELSE AT := hd ATStack;; ATStack := tl ATStack FI
```

$$can\text{-}interrupt'\ i \equiv i \in EIT\ -\ ATStack \wedge i \in interrupt\text{-}policy\ (hd\ ATStack)$$

SVC_now is used to directly switch to the SVC interrupt handler. $IRet$ returns control from an interrupt handler: it either switches control to svc_a (if svc_a has both been requested and is allowed to interrupt the *head of* $ATStack$) or returns control to the head of $ATStack$, which was saved as part of $ITake$. Although not explicitly part of *interleaving*, we require that the last command of svc_a_code, svc_s_code, and $handler\_code$ is $IRet$. This can be checked when instantiating the interleaving model to a specific system.

3.2 Instantiation to the *eChronos OS*

To model the *eChronos OS* we now just need to instantiate the above framework with the *OS* specific code. We give an overview of this instantiation below[7] while the full details can be found online [1] or in our previous paper [4].

$$eChronos\text{-}sys \equiv interleaving\ eChronos\text{-}init\ eChronos\text{-}svc_a\text{-}code\ eChronos\text{-}svc_s\text{-}code\ eChronos\text{-}handler\text{-}code\ eChronos\text{-}app\text{-}code$$

$$eChronos\text{-}svc_a\text{-}code \equiv schedule;;\ context\text{-}switch\ True;;\ IRet$$

$$eChronos\text{-}svc_s\text{-}code \equiv schedule;;\ context\text{-}switch\ False;;\ IRet$$

$$eChronos\text{-}handler\text{-}code\ i \equiv E :\in \{E' \mid E \subseteq E'\};;\ svc_aRequest;;\ IRet$$

```
$eChronos\text{-}app\text{-}code\ i \equiv userSyscall :\in \{SignalSend,\ Block\};;$
$IF\ userSyscall = SignalSend$
$THEN\ svc_aDisable;;\ R :\in \{R' \mid \forall i.\ R\ i = Some\ True \longrightarrow R'\ i = Some\ True\};;$
  $svc_aRequest;;\ svc_aEnable;;\ WHILE\ svc_aReq\ DO\ SKIP\ OD$
$ELSE\ IF\ userSyscall = Block$
  $THEN\ svc_aDisable;;\ R := R(i \mapsto False);;\ SVC\text{-}now;;\ svc_aEnable;;$
    $WHILE\ svc_aReq\ DO\ SKIP\ OD$
  $FI$
$FI$
```

In this work we focus on the scheduling behaviour, modelling only the parts that might affect scheduling decisions. These decisions depend on two variables, R for runnable tasks and E for the events signalled by interrupt handlers.

The parameters $eChronos\_svc_a\_code$ and $eChronos\_svc_s\_code$ are almost identical and are used by the *OS* to call the scheduler. First, *schedule*, defined below, picks the next task to run by first updating R through handling the unprocessed events E before using whichever scheduling policy is in place. After choosing the task a context switch is performed, with the old task being saved and the new task being placed on the stack.

[7] For presentation purposes, we omit ghost variables added to the program for verification purposes. The notation $x :\in S$ stands for non-deterministically updating x to be any element of S.

```
schedule ≡ nextT := None;;
WHILE nextT = None
DO E-tmp := E;; R := handle-events E-tmp R;; E := E − E-tmp;;
nextT := sched-policy R OD

context-switch preempt-enabled ≡
contexts := contexts(curUser ↦ (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
```

Next, *eChronos_handler_code* is mostly application-provided and is only allowed to affect the behaviour of the *OS* by expanding the set of events *E*. A flag is then raised saying that the scheduler should be run as soon as enabled. To finish, the handler, by calling *IRet*, either returns control to the previously executing context or, if allowed, switches control to the scheduler.

Finally, the only way the application code can affect the interleaving behaviour is via system calls. We model two representative syscalls, *signal_send* and *block*. In the *eChronos OS*, syscalls run with interrupts enabled, but preemption disabled; that is, they are surrounded by disabling and enabling the svc_a interrupt. This is to delay a call to the scheduler requested by an interrupt handler until *after* the *OS* syscall is finished. Each syscall ends with a loop that ensures that, if required, the scheduler executes before the *OS* returns control to the application. The syscall *signal_send* increases the set of runnable tasks and sets a flag indicating that the scheduler needs to be run, while *block* modifies *R* so that the specific application task is no longer runnable and then directly calls svc_s via *SVC_now*.

4 Proof framework and scheduler proof

4.1 Framework and compositionality lemma

In this section we explain our definition of *derivability of a (bare) parallel program c with respect to an invariant I, precondition p and postcondition q*, denoted $\|\!-_b\ I\ \ p\ \ c\ q$. We present the framework that we build to ease the proof of such a statement, by assuming helper invariants, and decomposing the proof into composable subproofs. In §4.2, we use this framework to state and prove the scheduler correctness $\|\!-_b\ \{\!|\mathit{scheduler\text{-}invariant}|\!\}\ \{\!|\mathit{True}|\!\}\ \mathit{eChronos\text{-}sys}\ \{\!|\mathit{False}|\!\}$.

We use the Owicki-Gries (*OG*) treatment of concurrency, captured in Isabelle/HOL by Prensa [20]. Reasoning about high-performance shared-variable system code requires a very low level of abstraction [4], and motivated our choosing *OG* over alternatives such as the the more structured Rely-Guarantee method. Futhermore, our goal was to verify existing code rather than to synthesise new code, i.e. a bottom-up proof rather than a top-down correctness-by-construction exercise. An attractive possibility, however, is to now use the invariants and assertions that *OG* and the code helped us to synthesise, and to explore whether with that "head start" a Rely-Guarantee approach would be possible: probably it would suggest proof-motivated modification to the code.

The *OG* method, introduced 40 years ago, extends the Hoare-style assertional-proof technique to reason about a number of individually sequential processes that are executed collectively in parallel. Namely, *OG* provides (1) a definition of *validity* of a Hoare triple over a (*fully annotated*) parallel composition of programs, denoted $\| \!= p \;\; c \; q$; (2) a set of proof rules for efficient verification of such a statement, with an associated *derivability* statement, denoted $\| \!- \; p \; c \; q$; (3) a *soundness* theorem of the rules w.r.t validity, namely $\| \!- \; p \; c \; q \longrightarrow \| \!= p \;\; c \; q$; and finally (4) an automated verification condition generator (VCG), i.e. a tactic *oghoare* in Isabelle/HOL to decompose a derivability statement into subgoals.

We explain these standard *OG* definitions before going into our extensions, which are proved sound with respect to the concurrency semantics. In the following, c and ac are mutually recursive datatypes; c is sequential code, which can contain a parallel composition of annotated code, ac. The parallel composition consists of a list of annotated programs with their postconditions. An annotated program can contain an AWAIT statement, whose body is a sequential program.

$$\begin{aligned}
c \equiv \;\; & x := v \mid c;;\ c \mid \mathit{IF}\ b\ \mathit{THEN}\ c\ \mathit{ELSE}\ c\ \mathit{FI} \mid \mathit{WHILE}\ b\ \ \mathit{DO}\ c\ \mathit{OD} \mid \\
& \mathit{COBEGIN}\ ts\ \mathit{COEND} \\
ts \equiv \;\; & [\,] \mid (aco,\ \{\!|a|\!\})\#ts \\
aco \equiv \;\; & \mathit{None} \mid \mathit{Some}\ ac \\
ac \equiv \;\; & \{\!|a|\!\}\ x := v \mid ac;;\ ac \mid \{\!|a|\!\}\ \mathit{IF}\ b\ \mathit{THEN}\ ac\ \mathit{ELSE}\ ac\ \mathit{FI} \mid \\
& \{\!|a|\!\}\ \mathit{WHILE}\ b\ \mathit{INV}\ \{\!|a|\!\}\ \mathit{DO}\ c\ \mathit{OD} \mid \{\!|a|\!\}\ \mathit{AWAIT}\ b\ \mathit{THEN}\ c\ \mathit{END}
\end{aligned}$$

In the above b is a boolean expression, and a is an assertion. Validity $\| \!= p \;\; c \; q$ is defined in terms of the execution semantics of the program, as in Hoare logic (all states reachable via multiple steps of execution from initial states satisfying the precondition will satisfy the postcondition). The execution of the standard language constructs is also defined as in Hoare logic. For parallel composition, one of the programs is at each step non-deterministically chosen to make progress. For the *AWAIT* statement, the body is executed, under the condition that the guard is satisfied (and that the body does not contain any parallel composition). The derivability rules ($\| \!- \; p \; c \; q$) are also the same as for Hoare logic. The key feature of *OG* is providing a proof rule for parallel composition, which consists in showing *local correctness* and *interference-freedom* for a list $[(\mathit{Some}\ ac_1, q_1), ..., (\mathit{Some}\ ac_n, q_n)]$ of annotated programs. Each program ac_i and postcondition q_i is first proved correct in isolation using standard sequential Hoare logic rules. Then, each assertion a in ac_i is proved to not be interfered with by any (annotated) statement $\{\!|a'|\!\}\ st'$ in another program ac_j (shown using standard Hoare logic as well: $\{\!|a \wedge a'|\!\}\ st'\ \{\!|a|\!\}$). This interference-freedom requirement makes the *OG* technique non-compositional and quadratic. However, in systems with limited concurrency like ours, the complexity is reduced and we apply proof engineering techniques to make it scale to verify real *OS* scheduling behavior.

Our first, small, extension to the original definition of derivability is to explicitly talk about the *invariant* of the program. The programs we target are infinite loops, where the postcondition is not reached. Therefore, their correctness can be expressed better in terms of an invariant over their execution. An invariant for an annotated program is merely a property repeated in all annotations. However, manually inserting it everywhere is tedious, error-prone and results in bad readability. Instead, we define the derivability of invariants as follows:

$$\parallel\!-_i\ I\ p\ c\ q \equiv \parallel\!-\ p\ (\mathit{add\text{-}inv\text{-}com}\ I\ c)\ q$$

where $\mathit{add\_inv\_com}\ I\ c$ simply inducts over the structure of program c and adds a conjunction with I to all annotations.

Our second extension is to be able to *assume* a helper invariant, while proving a main invariant. This feature is necessary in larger proofs where the property of interest relies on a number of other invariants. These invariants might need different sets of annotations; proving them all together quickly becomes unreadable, and even infeasible due to the explosion of complexity. It also makes it hard for multiple people to work on a single proof. We modify the original set of OG derivability rules to allow assuming an invariant, denoted $I\ \parallel\!-\ p\ c\ q$, as follows: preconditions get an extra conjunction with I (i.e. I can be assumed true initially) and postconditions get an extra implication from I (i.e. the postcondition itself only need to be proven if I holds). Then $\parallel\!-\ p\ c\ q$ simply stands for $\mathit{UNIV}\ \parallel\!-\ p\ c\ q$ (UNIV is the universal set) and $I'\ \parallel\!-_i\ I\ p\ c\ q$ stands for $I'\ \parallel\!-\ p\ (\mathit{add\text{-}inv\text{-}com}\ I\ c)\ q$. Putting things together, we provide a *compositionality lemma* to decompose the proof along the invariants.

$$\frac{I'\ \parallel\!-_i\ I\ p\ c\ q \qquad \parallel\!-_i\ I'\ p'\ c'\ q' \qquad \mathit{merge\text{-}prog\text{-}com}\ c\ c' = \mathit{Some}\ c''}{\parallel\!-_i\ (I' \cap I)\ (p \cap p')\ c''\ (q \cap q')} \tag{2}$$

where the merge of two programs requires the programs to only differ on annotations (i.e. have identical program text), and if so, returns the same program text with merged annotations (by conjunction). Our proof of the *eChronos* scheduler uses this lemma extensively, and would have not been tractable without it.

Finally we define the derivability of an invariant I over a *bare* program (i.e. not annotated) as the existence of an appropriate annotation sufficient to prove I, as follows:

$$\parallel\!-_b\ I\ \ p\ \ c\ q \equiv \exists\, c'.\ \mathit{extract\text{-}prg}\ c' = c \wedge \parallel\!-_i\ I\ p\ c'\ q \tag{3}$$

Since invariants are merely annotations, we can prove an introduction rule for derivability, which allows us to directly introduce helper invariants:

$$\frac{\exists\, c'.\ \mathit{extract\text{-}prg}\ c' = c \wedge \parallel\!-_i\ (I \cap I')\ p\ c'\ q}{\parallel\!-_b\ I\ \ p\ \ c\ q} \tag{4}$$

4.2 The statement and its proof

Now that we have defined our framework, we present the statement of *eChronos*' scheduler correctness:

$$\parallel\!-_b\ \{\!|\mathit{scheduler\text{-}invariant}|\!\}\ \{\!|\mathit{True}|\!\}\ \mathit{eChronos\text{-}sys}\ \{\!|\mathit{False}|\!\} \tag{1}$$

The definition of *eChronos_sys* is described in §3. Here we define *scheduler_invariant* and explain its proof.

As previously mentioned, the key property enforced by the *eChronos OS* is that the running application task is always the highest priority runnable task. We express this property as an invariant *scheduler_invariant*, defined as follows:

$$\begin{array}{l} \textit{scheduler-invariant}\ x \equiv \\ AT\ x \in U \land svc_a \in EIT\ x \land \neg\ svc_a Req\ x \longrightarrow \\ \textit{sched-policy}\ (\textit{handle-events}\ (E\ x)\ (R\ x)) = \textit{Some}\ (AT\ x) \end{array}$$

where x is the current state. The statement says that whenever the currently active task is a user (i.e. not an interrupt handler and not the scheduler), and we are not inside a system call (we will come back to that), then that user is indeed the one supposed to be running, according to the scheduling policy. The latter is expressed by the fact that the scheduling policy would choose the running user if re-run with the current values of events E and of the runnable set R.

The condition of not being in a system call is because, as explained in §3, preemption is turned off during system calls, meaning that any asynchronous request for the scheduler is delayed until the system call finishes running. Therefore, when the currently active task *is* a user, but is *inside* a system call, it might not be of highest priority. However, as soon as the system call is finished, the execution must *not* go back to that user but must instead immediately call the scheduler. The invariant should, therefore, only be checked outside of system calls. Being outside a system call is defined by the asynchronous scheduler being enabled. [8] The third premise represents the specific situation where preemption is turned back on, but the request for asynchronous scheduling is still on, waiting for the hardware to do the switch (as explained in §3). The execution only goes back to the user when this asynchronous scheduling request has been handled.

Now we describe how we prove (1). We use lemma (4), and for this we create a suitable complete annotation of *eChronos_sys* sufficient to prove the invariant *scheduler_invariant*. The details of the annotations are not particularly insightful, but the process of identifying them and incrementally building them is discussed at the end of this section. The main theorem we prove is:

$$\Vert\!-_i\ (\{\!\!\{\textit{scheduler-invariant}\}\!\!\} \cap \textit{helper-invs})\ \{\!\!\{\textit{True}\}\!\!\}\ \textit{eChronos-sys-ann}\ \{\!\!\{\textit{False}\}\!\!\} \qquad (5)$$

where *eChronos_sys_ann* is the fully annotated program, whose extracted program text is *eChronos_sys*, and where *helper_invs* are a set of nine invariants about *eChronos* state variables and data structures, required to prove *scheduler_invariant*. We prove lemma (5) by applying the compositionality lemma. We first prove the scheduler invariant *assuming* all the helper invariants:

$$\textit{helper-invs}\ \Vert\!-_i\ \{\!\!\{\textit{scheduler-invariant}\}\!\!\}\ \{\!\!\{\textit{True}\}\!\!\}\ \textit{eChronos-sys-ann}\ \{\!\!\{\textit{False}\}\!\!\} \qquad (6)$$

We then prove each helper invariant independently (and this can be done by different people, increasing efficiency). These invariants reveal much about the data structures but do not represent a high level correctness property of the

[8] Disabling the scheduler is one of the functions that the *eChronos OS* does *not* export, to keep control of latency, as mentioned in §1.

eChronos OS. We omit their definitions for space reasons (they are available online [1]), and just give two representative examples:

$$\begin{aligned}&\textit{last-stack-inv}\ x \equiv \textit{last}\ (\textit{AT}\ x\ \#\ \textit{ATStack}\ x) \in U\\&\textit{ghostP-inv}\ x \equiv \textit{ghostP}\ x \longrightarrow \textit{AT}\ x \in I \cup \{\textit{svc}_a,\ \textit{svc}_s\}\end{aligned}$$

The first invariant describes the allowed shape of the stack, namely that its last element is always a user task. It is representative of the invariants about the data structures. The second invariant is representative of the need for ghost variables to express where certain programs are in their execution. Here *ghostP* is a ghost flag that represents the fact that the asynchronous scheduler is running. The invariant *ghostP_inv* ensures that *ghostP* cannot be set if the active task is a user application. It is needed in the proofs of interference-freedom of user applications' assertions: it tells us that the asynchronous scheduler instructions cannot violate them as they cannot be running.

For each of the nine helper invariants, we prove that it is preserved by *eChronos_sys_ann*. Some of them rely on others so we reuse the compositionality lemma for these.

We proved all of these helper lemmas along with lemma (6) in an iterative process to discover the required annotations. Roughly, we start with minimal annotations, and run the *oghoare* tactic to generate the proof obligation for local correctness and interference-freedom. We apply the techniques discussed in §4.3 to reduce the number of subgoals by removing duplication and automatically discharging as many as possible. We are then left with a manageable set of subgoals, where we can identify which assertion in the program is being proved, and can start augmenting assertions as required to prove these subgoals.

4.3 Proof-engineering considerations

The *oghoare* proof tactic, offered in the Isabelle distribution and derived from [20], is the VCG used for decomposing an annotated program into subgoals. Each of these goals is ultimately either a judgement that the precondition for each program step is sufficient to demonstrate its postcondition or that a given annotation is not interfered with by anything else running in parallel.

The tactic is defined as a mutually recursive function that decomposes program sequencing, program (user-defined) annotations, and parallel composition. The provided implementation of this tactic results in a quadratic explosion of proof obligations: a ∼200 line parallel program takes *oghoare* ∼90 seconds to generate ∼3,000 subgoals.

Rather than solve each of these goals by hand, we chose to write a single custom tactic which was powerful enough to solve all of them. Here we leveraged Isabelle's existing proof automation and parallelisation infrastructure [17]. With some instrumentation and custom lemmas, Isabelle's *simplifier* [18, §3] can discharge almost all of the subgoals produced from *oghoare*. Isabelle's provided PARALLEL_GOALS tactical allows us to apply our custom tactic to all subgoals simultaneously in parallel, resulting in a significant reduction in overall proof processing time. Despite this infrastructure, however, these ∼3,000 subgoals can still take over an hour to prove. This is impractical from a proof engineering perspective, as this proof needs to be re-run every time the tactic is adjusted or the program annotations are changed. This prompted the development of several proof engineering methodologies that, although generally applicable, were instrumental in the completion of this proof.

Subgoal deduplication and memoization An initial investigation revealed that many of the proof obligations produced by *oghoare* were identical. Isabelle's provided *distinct_subgoals* tactic can remove duplicate subgoals, but takes over 30 seconds to complete on $3,000$ subgoals. We found that we could instead store proof obligations as *goal hypotheses* as they are produced, which are efficiently de-duplicated by Isabelle's proof kernel. This adds negligible overhead, and results in approximately a 3-fold reduction in the total number of proof obligations.

This large number of duplicate subgoals is a consequence of having many identical program annotations. The *oghoare* tactic recurses on the syntax of the annotated program, generating non-interference verification conditions for each annotation. Rather than complicate the implementation of *oghoare*, we chose to simply de-duplicate these proof obligations as they are produced.

Although this de-duplication reduces the total time required to finish the proof, it still indicates that the *oghoare* tactic is doing redundant computation and that the observed ∼90 second overhead could be reduced. To address this, we developed a new tactical for memoization, SUBGOAL_CACHE, which caches the result of applying a given tactic to the current subgoal. When the tactic is subsequently invoked again, the cache is consulted to determine if it contains a previously-computed result for the current subgoal. On a cache hit, the stored result is simply applied rather than having the tactic re-compute it. Isabelle's LCF-style proof kernel guarantees that such a cache is sound, as each cached result is a previously-checked subgoal that was produced by the kernel.

We applied SUBGOAL_CACHE to each of the mutually recursive tactics that *oghoare* comprises. Including subgoal de-duplication, this change reduces the running time of *oghoare* from ∼90 seconds to ∼5 seconds (on a ∼200 line program), without requiring any change to the underlying algorithm.

Subgoal proof skipping Even once these $\sim 3,000$ subgoals have been deduplicated down to $\sim 1,000$ distinct subgoals, discharging them all can take between 5 and 30 minutes, depending on the particular annotations. The development strategy was to run the simplifier on all the subgoals and then analyse those that remained unsolved. Each iteration required adding additional program annotations or providing the simplifier with additional lemmas in order to discharge more subgoals. This would then require waiting for up to 30 minutes again to see if the change was successful.

To save time, we added another tactical, PARALLEL_GOALS_SKIP, which builds on Isabelle's *skip_proofs* mode and PARALLEL_GOALS tactical. This tactical is equivalent to the existing PARALLEL_GOALS, but records which subgoals were successfully discharged as global state data. This global record can then be accessed if the tactic is re-executed in Isabelle/jEdit (after, for example, going back and adding another annotation to the function). When **re**-executing the tactic, subgoals that were previously discharged are instead simply *skipped* and assumed solved. In practice, this reduces the effective iteration time from minutes to seconds, depending on the significance of the change. When the proof is complete, PARALLEL_GOALS_SKIP is then replaced with PARALLEL_GOALS in order to avoid skipping proofs and guarantee soundness.

Together these methodologies make this approach far more tractable and scalable than was previously thought possible.

5 Related work

We discuss models of interrupts, verification of operating systems, models of real-world systems with concurrency, and automation and mechanisation of *OG*.

The closest work to ours formalises interrupts explicitly [9,13,12], using "ownership" to reason about resource sharing. That provides verification modularity, but the run-time discipline it induces limits the effective concurrency unacceptably for a real-time system where low latency is paramount. Indeed, they assume that interrupts are disabled when data is shared and during scheduling and context switching; we do not. They also do not support nested interrupts, although some [9] do suggest how they could. However, one [12] does support multicore.

Other works in *OS* verification, less closely related, either do not model interrupts, or target systems where *OS* code runs with interrupts disabled. Close to the *eChronos OS* is FreeRTOS [2], a real-time *OS* for embedded microcontrollers. Its verification has been the target of several projects: in [7,10,6], the focus of the verification is on the scheduling policy itself (picking the next task), or on the correct handling of the data-structure lists and tasks by the scheduler. While FreeRTOS runs with interrupts mostly enabled, interrupt handling is not modelled in these works, nor is context-switching. That work is complementary to ours, where we leave the policy generic. In [5], the authors target progress properties (absence of data-race and deadlock) of their proposed multicore version of FreeRTOS. Again, this is orthogonal to our focus on correctness. Another embedded real-time *OS* that has been verified [14] is used in the OSEK/VDX automotive standard. It has been model-checked in CSP, and the interleaving model has some similarities with ours, where tasks are in parallel composition with the scheduler. But again, interrupts are out of scope.

In *OS* verification generally, existing, larger *OS*-es that are formally verified [16,21], run with interrupts disabled throughout the executions of system calls from applications, making those calls' executions sequential.

Finally, a notable verification effort outside the pure *OS* world is on-the-fly garbage collection (GC) in a relaxed memory model [11]. They too chose the rigour of Isabelle/HOL, and used a system-wide invariant. Concurrency control is via message passing, while the *eChronos OS* uses shared variables.

A GC is also the target of the main existing use of formalised and mechanised Owicki-Gries. Prensa, the author of the *OG* formalisation [20] in Isabelle/HOL, on which our work is based, used her framework to verify a simple GC algorithm. In terms of scale, Prensa's model contains only two threads in parallel, one of which contains only 2–3 instructions: this generates only ∼100 verification conditions. Our proof effort generates ∼$3,000$ *VC*'s, and so requires significant proof engineering [15] to be feasible. Also, Prensa's work does not extract the correctness property in a separate, well-identified invariant annotation. Nor does it allow control of concurrency and interleaving between the parallel processes, that is, the inclusion of a task-scheduler which is itself subject to verification.

6 Conclusion

Our contribution has been the intersection of three ideas: that modern proof-automation now makes Owicki-Gries reasoning about concurrency feasible for much larger programs than before; that *OG* can be used in a style that allows reasoning about programs that control and limit their own concurrency; and that an ideal target to test these ideas is a small, highly interleaved preemptive real-time operating system. To our knowledge this is the first proof of an *OS* system running with interrupts enabled even during scheduling, and allowing nested interrupts. The proof does make assumptions about application code conventions, as remarked in §1, precisely because the *OS* is not hardware protected. But these are statically checkable and reasonable for applications running on a real-time *OS*. Our experience in doing this proof should be useful to the wider ITP community: we can contribute proof-engineering insights for dealing with a huge amount of goals. Furthermore, our proof of scheduler correctness for a real-time *OS* already in commercial use in medical devices has real, practical value.

The work we have done so far sits roughly in the middle of a complete verification of an application running on the *eChronos OS* platform. Above, further work could provide a verified *OS* API specification that application programmers could use to prove their programs' correct behavior. Below, we have yet to prove refinement between the large-grained atomic steps and the low-level primitives for concurrency, and that the *OG* model on which our proof is based accurately captures the behaviour of our target processor. Those last two will be our next step, as well as continuing to develop proof-engineering techniques crucially needed for efficient and scalable concurrency software verification.

Acknowledgements The authors would like to thanks Gerwin Klein and Stefan Götz for their feedback on drafts of this paper. NICTA is funded by the Australian Government through the Department of Communications and by the Australian Research Council through the ICT Centre-of-Excellence Program.

References

1. eChronos model and proofs, `https://github.com/echronos/echronos-proofs`
2. FreeRTOS, `http://www.freertos.org/`
3. The eChronos OS, `http://echronos.systems`

4. Andronick, J., Lewis, C., Morgan, C.: Controlled Owicki-Gries concurrency: Reasoning about the preemptible eChronos embedded operating system. In: Workshop on Models for Formal Analysis of Real Systems (MARS) (2015)
5. Chandrasekaran, P., Kumar, K.B.S., Minz, R.L., D'Souza, D., Meshram, L.: A multi-core version of FreeRTOS verified for datarace and deadlock freedom. In: MEMOCODE. pp. 62–71. IEEE (2014)
6. Cheng, S., Woodcock, J., D'Souza, D.: Using formal reasoning on a model of tasks for FreeRTOS. Formal Aspects of Computing 27(1), 167–192 (2015)
7. Divakaran, S., D'Souza, D., Kushwah, A., Sampath, P., Sridhar, N., Woodcock, J.: Refinement-based verification of the FreeRTOS scheduler in VCC. In: Butler, M., Conchon, S., Zadi, F. (eds.) ICFEM. Lecture Notes in Computer Science, vol. 9407, pp. 170–186. Springer (2015)
8. Feijen, W.H.J., van Gasteren, A.J.M.: On a Method of Multiprogramming. Monographs in Computer Science, Springer (1999)
9. Feng, X., Shao, Z., Guo, Y., Dong, Y.: Certifying low-level programs with hardware interrupts and preemptive threads. Journal of Automated Reasoning 42(2-4), 301–347 (2009)
10. Ferreira, J.F., Gherghina, C., He, G., Qin, S., Chin, W.N.: Automated verification of the FreeRTOS scheduler in HIP/SLEEK. International Journal on Software Tools for Technology Transfer 16(4), 381–397 (2014)
11. Gammie, P., Hosking, T.A., Engelhardt, K.: Relaxing safely: Verified on-the-fly garbage collection for x86-TSO. In: Steve Blackburn (ed.) PLDI 2015: the 36th annual ACM SIGPLAN conference on Programming Language Design and Implementation. p. 11. ACM (2015)
12. Gotsman, A., Yang, H.: Modular verification of preemptive OS kernels. J. Funct. Program. 23(4), 452–514 (2013)
13. Guo, Y., Zhang, H.: Verifying preemptive kernel code with preemption control support. In: 2014 Theoretical Aspects of Software Engineering Conference, TASE 2014, Changsha, China, September 1-3, 2014. pp. 26–33. IEEE (2014)
14. Huang, Y., Zhao, Y., Zhu, L., Li, Q., Zhu, H., Shi, J.: Modeling and verifying the code-level OSEK/VDX operating system with CSP. In: Theoretical Aspects of Software Engineering (TASE). pp. 142–149. IEEE (2011)
15. Klein, G.: Proof engineering considered essential. In: Cliff Jones, Pekka Pihlajasaari, Jun Sun (ed.) FM. pp. 16–21. Springer (2014)
16. Klein, G., Andronick, J., Elphinstone, K., Murray, T., Sewell, T., Kolanski, R., Heiser, G.: Comprehensive formal verification of an OS microkernel. Trans. Comp. Syst. 32(1), 2:1–2:70 (2014)
17. Matthews, D.C., Wenzel, M.: Efficient parallel programming in Poly/ML and Isabelle/ML. In: Petersen, L., Pontelli, E. (eds.) POPL 2010 WS Declarative Aspects Multicore Progr. pp. 53–62. ACM (2010)
18. Nipkow, T., Paulson, L., Wenzel, M.: Isabelle/HOL — A Proof Assistant for Higher-Order Logic, LNCS, vol. 2283. Springer (2002)
19. Owicki, S., Gries, D.: An axiomatic proof technique for parallel programs. Acta Informatica 6, 319–340 (1976)
20. Prensa Nieto, L.: Verification of parallel programs with the Owicki-Gries and rely-guarantee methods in Isabelle/HOL. Ph.D. thesis, T.U. München (2002)
21. Yang, J., Hawblitzel, C.: Safe to the last instruction: automated verification of a type-safe operating system. In: 2010 PLDI. pp. 99–110. ACM (2010)