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

MARS 2015

June Andronick, Corey Lewis, Carroll Morgan
November 2015
Aim

Formal model of the **eChronos** embedded RT OS which requires **Concurrency** reasoning for which we use **Owicki-Gries** method.

- The challenge is to reason about the real system.
- The approach is to use the Owicki-Gries method.
eChronos

What: Small real-time OS library (~500 SLOC)
(Joint development with Breakaway Consulting)

Where: Embedded devices, with limited resources, no memory-protection

Job:
- provides synchronisation primitives
- schedules tasks according to priorities

“the running task must be of highest-priority”

applications

- task A
- task B

can call

C=current
R=runnable
B=blocked

signal_send
signal_wait

...
**eChronos**

What: Small real-time OS library (~500 SLOC)  
(Joint development with Breakaway Consulting)

Where: Embedded devices, with limited resources, no memory-protection

Job:  
- provides synchronisation primitives  
- schedules tasks according to priorities

---

**API**

- `signal_send`
- `signal_wait`
- `...`

**Internal functions**

- `interrupt 1`
- `interrupt 2`
- `interrupt 3`

**Applications**

- task A
- task B
- handler 1
- handler 2
- handler 3

---

**triggered by**

- interrupt 1
- interrupt 2
- interrupt 3

**can call**

- can call
- can call (some)
eChronos

Characteristics:  
- small, fast

Aim:  
- verified

Challenges:  
- tasks can be preempted by another task
- OS can be interrupted by external event

Our approach applies to OS systems with these characteristics:
- tasks can be preempted by higher priority tasks
- OS code can be interrupted by external event
Aim

Formal model of the *eChronos* embedded RT OS which requires *Concurrency* reasoning for which we use *Owicki-Gries* method.

The real system

The challenge

The approach
Owicki-Gries

What: Extension of Hoare logic to shared-variable parallel programs
(Suzanne Owicki and David Gries, 1976)

\[ \text{cobegin } S_1 \parallel S_2 \parallel \ldots \parallel S_n \text{ coend} \]
\[ \text{await } B \text{ then } S \]

Why: Small, fast
Owicki-Gries

How:

Program P

\{is\_odd \ x\}\n\x:=\x+1;  
\{is\_even \ x\}\n
Program P'

\x:=\x+2;

P is (sequentially) correct  
Local correctness

P' does not interfere with P  
Interference freedom

P is globally correct
Owicki-Gries

From a parallel composition of fully annotated programs, generates correctness verification conditions

- **Local correctness**: prove each \( \{a_i\} c_i \{a_{i+1}\} \)
- **Interference freedom**: for each assertion \( a \) in \( P \), and each command \( c' \) in \( P' \), prove that \( \{a \land a'\} c' \{a\} \)

❗ Quadratic explosion of proof obligations, not compositional

⇒ For our system, interleaving is more controlled

+ Use automation power of modern theorem provers

Leonor Prensa Nieto 2002
Aim:

- use OG to model interleaving between eChronos code, interrupt code, and tasks

Challenges:

- tasks are not 1st class citizens
- concurrency is uncontrolled
Contributions

Formal model of **eChronos** using **Owicki-Gries** method

**Challenges:**
- tasks can be preempted
- OS can be interrupted

**Challenges:**
- tasks are not 1<sup>st</sup> class citizens
- concurrency is uncontrolled

**Contributions**

1. Tasks as 1<sup>st</sup> class citizens in O-G
   “AWAIT-painting”

2. Controlled interleaving model
   *Hardware API model*

3. Formal model of eChronos
   *to prove that the running task is always the highest-priority runnable task*

Formalised in Isabelle/HOL theorem prover
Tasks as 1\textsuperscript{st} class citizens in O-G

“AWAIT-painting”

1.

Controlled interleaving model

\textit{Hardware API model}

2.

Formal model of eChronos

\textit{to prove that the running task is always the highest-priority runnable task}

3.
Context switching between tasks

No arbitrary concurrency between all these programs
Can only switch from A to B if B become the active task
Context switching between tasks

➡️ We introduce: Variable AT (Active Task)

➡️ We “AWAIT-paint” all statements:

```plaintext
AWAIT AT=A THEN a_1;
AWAIT AT=A THEN a_2;
AWAIT AT=A THEN a_3;
AWAIT AT=B THEN b_1;
AWAIT AT=B THEN b_2;
AWAIT AT=B THEN b_3;
```

➡️ We automate this with an “await_paint task-id code” command:

```plaintext
await_paint A code_A;
```

➡️ We model context switch: `context_switch task_id ≡ AT:=task_id;`

14 Controlled OG Concurrency: Reasoning about the Preemptible eChronos Embedded OS | June Andronick, Corey Lewis, Carroll Morgan
Tasks as 1st class citizens

Owicky-Gries
1975
Leonor Prensa Nieto
2002

AT
await_paint
context_switch
1. Tasks as 1st class citizens in O-G
   “AWAIT-painting”

2. Controlled interleaving model
   Hardware API model

3. Formal model of eChronos
   to prove that the running task is always
   the highest-priority runnable task
Recall eChronos

![Diagram showing eChronos applications and internal functions]

- **Applications**
  - task A
  - task B

- **Internal Functions**
  - handler 1
  - handler 2
  - handler 3
  - interrupt 1
  - interrupt 2
  - interrupt 3

- **API**
  - signal_send
  - signal_wait
  - yield
  - start
  - ...
Execution model

A task runs until:

1. **calling** a OS API function that changes runnable tasks

* Note: no mode switch between OS and task in such constrained hardware.
Execution model

A task runs until:

1. **calling** a OS API function that changes runnable tasks

- **Note:** no mode switch between OS and task in such constrained hardware.

** Note: using ARM “supervisor call” (svc) mechanism
Execution model

A task runs until:

1. **Calling** a OS API function that changes runnable tasks

2. Being **interrupted** by a handler that changes runnable tasks
A task runs until:

3. **calling** a OS API function that gets **interrupted** by a handler that changes runnable tasks

Our approach applies to OS systems with these characteristics:
- tasks can be preempted by higher priority tasks
- OS code can be interrupted by external event
Controlled Interleaving in OG

We do not have arbitrary concurrency between all these programs.

Other interleaving is controlled by hardware instructions:

- context-switch
- return-from-interrupt
- interrupt masking
Modelling application tasks

WHILE True DO
  await_paint A code_A
END

WHILE True DO
  await_paint B code_B
END

Interleaving can only happen if one instruction is a call to an OS function calling the scheduler calling `context_switch`... or if an interrupt happens
Modelling application tasks

<table>
<thead>
<tr>
<th>task A₁</th>
<th>...</th>
<th>task Aₙ</th>
</tr>
</thead>
<tbody>
<tr>
<td>code_A₁</td>
<td>...</td>
<td>code_Aₙ</td>
</tr>
</tbody>
</table>

⇒ We can generalise to \( n \) tasks

SCHEME \([0 \leq i < n]\)

WHILE True DO
    await_paint \( A_i \) code_Aᵢ
END
Modelling interrupt handlers

SCHEME \([0 \leq i < n]\)
\[\text{WHILE True DO}\]
\text{task A} \quad \text{task B} \quad \text{scheduler} \quad \text{handler 1}

SCHEME \([0 \leq j < m]\)
\[\text{WHILE True DO}\]
\text{ITake}(H_j)
\text{await_paint} H_j \text{ code}_h_j
\text{IRet()}
\text{END}

Controlled OG Concurrency: Reasoning about the Preemptible eChronos Embedded OS | June Andronick, Corey Lewis, Carroll Morgan
Modelling interrupt handlers

\[
\begin{align*}
\text{task A}_1 & \quad | \quad \ldots \quad | \quad \text{task A}_n \\
\text{code}_{A_1} & | \quad \ldots \quad | \quad \text{code}_{A_n}
\end{align*}
\]

\[
\begin{align*}
\text{handler H}_1 & \quad | \quad \ldots \quad | \quad \text{handler H}_m \\
\text{code}_{h_1} & | \quad \ldots \quad | \quad \text{code}_{h_m}
\end{align*}
\]

\[
\text{ITake}(X) \equiv \text{AT} := X
\]

\[
\text{SCHEME}\ [0 \leq j < m] \\
\text{WHILE}\ \text{True}\ \text{DO} \\
\quad \underline{\text{ITake}(H_j)} \\
\quad \text{await\_paint}\ H_j\ \text{code}_{h_j} \\
\quad \text{await\_paint}\ H_j\ \text{IREt()} \\
\text{END}
\]
Modelling interrupt handlers

\[
\text{SCHEME } [0 \leq j < m]\n\] 
\[
\text{WHILE True DO}
\]
\[
\text{ITake(H}_j\text{)}
\]
\[
\text{await\_paint } H_j \text{ code}_h_j
\]
\[
\text{await\_paint } H_j \text{ IRet()}
\]
\[
\text{END}
\]

- What if X is masked?
- New variable \( EI \) (Enabled Interrupts)
- New hardware functions
  - \( \text{IntDisable}(X) \equiv EI := EI - X \)
  - \( \text{IntEnable}(X) \equiv EI := EI \cup X \)
Modelling interrupt handlers

\[
\text{ITake}(X) \equiv \text{AWAIT } X \in EI \\
\text{THEN} \\
\text{push } \text{AT} \text{ ATstack; } \\
\text{AT} := X
\]

\[
\text{SCHEME } [0 \leq j < m] \\
\text{WHILE True DO} \\
\text{ITake}(H_j) \\
\text{await_paint } H_j \text{ code}_h_j \\
\text{await_paint } H_j \text{ IRet()} \\
\text{END}
\]

- How to return to previously running task eventually?
- New variable ATstack
Modelling interrupt handlers

\[
\text{SCHEME [0} \leq j \text{< m]}
\]

\[
\begin{align*}
\text{WHILE } \text{True DO} \\
\text{ITake}(H_j) \\
\text{await} \_\text{paint } H_j \\
\text{IRet}() \\
\text{END}
\end{align*}
\]

\[
IRet() \equiv \ AT := \text{sched};
\]
Modelling interrupt handlers

\[\text{SCHEME } [0 \leq j < m] \]
\[
\text{WHILE True DO}
\]
\[
\text{ITake}(H_j)
\]
\[
\text{await_paint } H_j \text{ code}_h_j
\]
\[
\text{await_paint } H_j \text{ IRet()}
\]
\[
\text{END}
\]

\[\text{IRet()} \equiv \text{AT} := \text{sched};\]

- New hardware functions
  - SchedDisable() ≡ EI:=EI−sched
  - SchedEnable() ≡ EI:=EI∪sched

- New flag schedReq
Modelling interrupt handlers

\[
\begin{align*}
\text{IRet}() & \equiv \ \text{AT} := \text{sched}; \\
\end{align*}
\]

SCHEME $[0 \leq j < m]$

\[
\begin{align*}
\text{WHILE True DO} & \text{ }
\text{ITake}(H_j) \\
\text{await\_paint } H_j & \text{code}_h_j \\
\text{await\_paint } H_j & \text{IRet}() \\
\text{END}
\end{align*}
\]

⇒ New hardware functions
\text{SchedDisable()} \equiv \text{EI} := \text{EI} - \text{sched}
\text{SchedEnable()} \equiv \text{EI} := \text{EI} \cup \text{sched}

⇒ New flag \text{schedReq}
Modelling interrupt handlers

\begin{align*}
&\text{code}_{A_1} \mid \cdots \mid \text{code}_{A_n} \\
&\text{code}_{h_1} \mid \cdots \mid \text{code}_{h_m}
\end{align*}

\begin{align*}
\text{IRet}() &= \\
\text{IF } \text{schedReq} \land \text{sched} \in E_I \\
\text{THEN } \text{AT} := \text{sched} \\
\text{ELSE } \text{AT} := \text{pop ATstack}
\end{align*}

\begin{align*}
\text{SCHEME } [0 \leq j < m] \\
\text{WHILE } \text{True DO} \\
\quad \text{ITake(H}_j\text{)} \\
\quad \text{await\_paint H}_j \text{ code}_{h_j} \\
\quad \text{await\_paint H}_j \text{ IRet()} \\
\text{END}
\end{align*}
Modelling interrupt handlers

```
WHILE True DO *
   ITakeSched()
   await_paint sched code_sch
   await_paint sched IRet()
END
```

```
ITakeSched() ≡
   AWAIT schedReq ∧ sched ∈ EI
   THEN
       schedReq:=False;
       push AT ATStack;
       AT:=shed;
   END
```

* simplified version, full details in paper
Our model of interleaving and HW API

Variables $AT, ATStack, EI, schedReq$

+ 
  $\text{IntEnable}(X)$
  $\text{IntDisable}(X)$
  $\text{SchedEnable}()$
  $\text{SchedDisable}()$

  $\text{ITake}(X)$
  $\text{IRet}()$
  $\text{ITakeSched}()$

SCHEME $[0 \leq i < n]$

WHILE True DO
  $\text{ITakeSched}()$
  $\text{await_paint sched code_sch}$
  $\text{await_paint sched IRet()}$
END

SCHEME $[0 \leq j < m]$

WHILE True DO
  $\text{ITake}(H_j)$
  $\text{await_paint H_j code_h_j}$
  $\text{await_paint H_j IRet()}$
END

...
1. Tasks as 1\textsuperscript{st} class citizens in O-G
   “AWAIT-painting”

2. Controlled interleaving model
   \textit{Hardware API model}

3. Formal model of eChronos
   \textit{to prove that the running task is always the highest-priority runnable task}
Instantiation to eChronos

```
<table>
<thead>
<tr>
<th>task A_1</th>
<th>...</th>
<th>task A_n</th>
<th>scheduler</th>
<th>...</th>
<th>handler H_1</th>
<th>...</th>
<th>handler H_m</th>
</tr>
</thead>
</table>

SCHEME [0 ≤ i < n]
WHILE True DO
  ITakeSched()
  await-paint A_i
  code-A_i
END

code_A_i ≡
SchedDisable();
R := changeRunnable(R);
SchedEnable();

SCHEME [0 ≤ j < m]
WHILE True DO
  ITake(H_j)
  await-paint H_j
  IRet()
  await-paint H_j
  code_h_j
END

code-sch ≡
nextT := None;
WHILE nextT = None
  DO
    Etmp := E;
    R := handleEvents Etmp R;
    E := E - Etmp;
    nextT := schedPolicy(R);
  OD;
  context_switch(nextT);
```

SCHEME [0 ≤ j < m]
WHILE True DO
  ITake(H_j)
  await-paint H_j
  IRet()
  await-paint H_j
  code_h_j
END

code-h_i ≡
E := changeEvents();
schedReq := True;

...
Summary

1. Tasks as 1st class citizens in O-G
   “AWAIT-painting”

2. Controlled interleaving model
   Hardware API model

3. Formal model of eChronos
   to prove that the running task is always
   the highest-priority runnable task

```plaintext
await_paint task_id code
AT, ATStack, EI, schedReq
  IntEnable(X)
  IntDisable(X)
  SchedEnable()
  SchedDisable()
  ITake(X)
  IRet()
  ITakeSched()

code_A_i ≡ ...
code_h_i ≡ ...
code_sch ≡ ...
```
Future work

Tasks as 1st class citizens in O-G
“AWAIT-painting”

Controlled interleaving model
Hardware API model

Formal model of eChronos
\textit{to prove that the running task is always the highest-priority runnable task}

Prove this property holds on our eChronos model

Prove the model correctly abstracts eChronos code
Informal arguments of validity

AWAIT statements do not exist in the implementation!
⇒ only use them to represent atomicity enforced by hardware

Introduced variables do no exist in the implementation!
⇒ only modified by hardware API functions
Preliminary arguments of practicality

We have started proving the correctness of the scheduling behaviour

- proof done* for an initial version of the model
- ∼10,000 verification conditions generated
- down to ∼500 by removing redundant conditions automatically
- down to ∼10 after automatic discharge by Isabelle/HOL

*almost ;-)
Thank you