Self-Verifying Concurrent Programming

Peter Welch¹, Matt Pedersen², Fred Barnes¹ and Carl Ritson²

¹School of Computing, University of Kent, UK
²School of Computer Science, UNLV, USA

IFIP WG2.4, Cape May, 29th. September, 2011

15-Oct-11 Copyleft (GPL) P.H.Welch and J.B.Pedersen

Self-Verifying Concurrent Programming

Aim:
To enable formal verification of $\text{occam-π}$ programs to be conducted within the language itself ... as a matter of course by the programmer.

How:
Extend $\text{occam-π}$ with verification qualifiers and assertions. Modify the compiler to generate (minimal) CSP code from programs using these qualifiers and assertions, bounce this off the FDR model checker and report back in terms of the source code.

occam-π, the process algebra

Why?
Because, now, we can ... it’s needed ... and it’s time!

occam-π, the process algebra

Example: autonomous robot component

The following example has been developed from one first worked through in a single lesson of a graduate class in concurrency at UNLV in the spring of 2010.

Example: autonomous robot component

Device : real-time controller for 8 channels (4 input, 4 output).

Device : real-time controller for 8 channels (4 input, 4 output).

Device : real-time controller for 8 channels (4 input, 4 output).

Example: autonomous robot component

Device : real-time controller for 8 channels (4 input, 4 output).

There are 3 sub-components: $P_0$ (weapons systems), $P_1$ (vision processing) and $P_2$ (motion stabiliser).

They exchange information over internal channels (ask, ans, ping) and coordinate actions with an internal barrier (bar).
Example: autonomous robot component

They exchange information over internal channels (ask, ans, ping) and coordinate actions with an internal barrier.

CSP semantics apply. Channel communication is unbuffered (sender waits for receiver and vice-versa). Any process reaching a barrier waits for all processes to reach the barrier.

Example: autonomous robot component

Behaviour: two representations

cocom: for compiling to a runnable system.
(memory overheads <= 32 bytes per process / synchronisation overheads of order tens of nanoseconds / eats multicore nodes for breakfast.)

CSP: for formal analysis.
[FDR2 model checker + other (simple) formal reasoning.]

Behaviour: one representation

cocom: for compiling to a runnable system.
(memory overheads <= 32 bytes per process / synchronisation overheads of order tens of nanoseconds / eats multicore nodes for breakfast.)

cocom: for formal analysis.
[verify qualifiers and (FDR) assertions + other (simple) formal reasoning.]

Behaviour: what are we looking for?

deadlock: might it ever stop? (e.g. P0 and P2 want to synchronise on bar, but P1 wants to ping.)

live: might it get busy... but refuse all external signals? (e.g. P0, P1 and P2 start engaging in an infinite sequence of internal channel or barrier synchronisations on ask, ans, ping and bar.)

Behaviour: what are we looking for?

safety: might it ever engage in an incorrect sequence of external signals?

liveness: will it engage in correct sequences of external signals, as required?
For the behaviour analysis in this example, data values and computations are not relevant. For simplicity, they are omitted in these codes, with all message content abstracted to zero.

Informal Intuitive

Device

P0

P1

P2

P3

WHILE TRUE

SEQ

INT x, y, z:

WHILE TRUE

SEQ

INT x, y, z:

WHILE TRUE

SEQ

INT x, y, z:

WHILE TRUE

SEQ

INT x, y, z:

What patterns of external (blue) signalling are possible from Device?
Informal

Intuitive

WHILE TRUE
INT x, y, z:
SEQ
INT x, y, z:
WHILE TRUE
SEQ
INT x, y, z:

If $b_0$ second, then?

If $b_0$ second, then?

If $a_0$, $b_0$, $a_1$ second, then?

If $a_0$, $b_0$, $a_1$ second, then?

What’s second?

$b_0$ or $a_1$

$a_1$ then $b_1$

$a_1$ then $b_1$

$a_0$, $b_0$, $a_1$, $b_1$
Informal

```
WHILE TRUE
SEQ
INT x: INT x:
```

If \( a_0, a_1, b_1, b_0 \) then?
\( b_0 \) and \( b_1 \)
\(<a_0, a_1>\)
(* any order)

Informal

```
WHILE TRUE
SEQ
INT x, y, z: INT x, y, z:
```

If \( a_1 \) second, then?
\( b_0 \) and \( b_1 \)
\(<a_0, a_1, b_0, b_1>\)
\(<a_0, a_1, b_1, b_0>\)

Informal

```
WHILE TRUE
SEQ
INT x, y, z: INT x, y, z:
```

If \( a_1 \) second, then?
\( b_0 \) and \( b_1 \)
\(<a_0, a_1, b_0, b_1>\)
\(<a_0, a_1, b_1, b_0>\)

Informal

```
WHILE TRUE
SEQ
INT x, y, z: INT x, y, z:
```

If \( a_1 \) second, then?
\( b_0 \) and \( b_1 \)
\(<a_0, a_1, b_0, b_1>\)
\(<a_0, a_1, b_1, b_0>\)

Behaviour: \( \text{occam-}\pi \)
\( \pi \)
\( \pi \)
\( \pi \)
\( \pi \)

\begin{tabular}{|l|l|}
\hline
If \( a_0, a_1, b_0, b_1 \) then? & \( c_0 \) and \( \pi \) \\
\hline
If \( a_0, a_1, b_1, b_0 \) then? & \( c_0 \) and \( \pi \) \\
\hline
If \( a_0, a_1 \) then? & \( c_0 \) and \( d_0 \) \\
\hline
\end{tabular}

That's 18 possible orderings of the first 7 signals.

What happens when the sub-processes start looping?
Formal

Verify Qualifiers: data

If we generated CSP$_A$, that fully reflected the semantics of the occam-$\pi$ source code, we would quickly produce a system with too many states for any feasible model checking. For instance, a single INT variable has 4G possible states!

By default, therefore, data values are ignored when generating the CSP$_B$. For instance:

\[
\text{PROC P (VAL INT i, CHAN INT c!, d!)}
\]
\[
\begin{array}{c}
\text{if } i = 42 \\
\text{TRUE} \\
\text{d} ! i
\end{array}
\]

maps just to:

\[
\begin{align*}
F (c) &= c \rightarrow \text{SKIP} \\
Q (i, c, d) &= \text{if } i = 42 \text{ then } c \rightarrow \text{SKIP else } d \rightarrow \text{SKIP}
\end{align*}
\]

If data values are significant, we qualify their types:

\[
\text{PROC Q (VAL VERIFY INT i, CHAN INT c!, d!)}
\]
\[
\begin{array}{c}
\text{if } i = 42 \\
\text{TRUE} \\
\text{d} ! i
\end{array}
\]

Such data variables are tracked and the above now maps to:

\[
\begin{align*}
Q (i, c, d) &= \text{if } i = 42 \text{ then } c \rightarrow \text{SKIP else } d \rightarrow \text{SKIP}
\end{align*}
\]
**Verify Assertions: occam-π**

Without testing the system, we can assert straight away that our `Device` is deterministic and free from deadlock and livelock – and that it doesn’t terminate.

**Verify Qualifiers: processes**

To verify behaviours beyond determinism, deadlock and livelock freedom and termination, we need some way to express the behaviours we want. We can use `occam-π` for this, together with refinement.

**Process Refinement**

A trace of a process is a finite sequence of external events that it can perform.

Process `P` `trace refines` `Q` if all traces of `P` are traces of `Q`.

**Behaviour: occam-π (verifyable)**

To check whether particular event sequences (traces) may initially be performed by `Device` ... e.g.

Define processes that have no choice in the matter ... e.g.
Safety

Define processes that have no choice in the matter... e.g.

\[
\begin{align*}
\text{VERIFY T0 REFINES T Device} & \quad \checkmark \quad \text{... which verifies our intuition}.
\end{align*}
\]

Safety

Define processes that have no choice in the matter... e.g.

\[
\begin{align*}
\text{VERIFY T1 REFINES T Device} & \quad \times \quad \text{... which verifies our intuition}.
\end{align*}
\]

Informal understanding

Let's ask a more difficult question about the continuous running of the system. Suppose the robot would do something very bad if its controller Device were ever to perform a signal twice on \(d0\) without a signal on \(d1\) in between. Might this ever happen?

Simple: write a process that checks all signals to/from Device, looking for the bad scenario and deliberately deadlocking the monitored system if spotted. This is just programming...

Informal understanding

Simple: write a process that checks all signals to/from Device, looking for the bad scenario and deliberately deadlocking the monitored system if spotted. This is just programming...

P2P2

Informal understanding

Simple: write a process that checks all signals to/from Device, looking for the bad scenario and deliberately deadlocking the monitored system if spotted. This is just programming...

Informal understanding

Simple: write a process that checks all signals to/from Device, looking for the bad scenario and deliberately deadlocking the monitored system if spotted. This is just programming...

Informal understanding

Simple: write a process that checks all signals to/from Device, looking for the bad scenario and deliberately deadlocking the monitored system if spotted. This is just programming...
Simple: write a process that checks all signals to/from monitored system if spotted. This is just programming.

**Behaviour:**

```
VERIFY PROC Check (CHAN INT a0, b0, c0, d0, d1, alive1)
  INT n := 0
  WHILE TRUE
    SEQ
    a0 ! 0
    IF n >= 2
      STOP -- refuse all further signals (forcing deadlock)
    THEN
      ... process next signal (maintain n)
  END
END
```

Trace refinement:

```
VERIY PROC CheckDevice (CHAN INT alive)
  CHECK (a0?, b0?, c0!, a1?, b1?, c1!, d0!, d1!, alive)
  Device (a0?, b0?, c0!, a1?, b1?, c1!, d0!, d1!) DEVICE
  Check (a0!, b0!, c0?, a1!, b1!, c1?, d0?, d1?, alive)
  Verify (n, alives)
  return
END
```

Note: protocol checking monitors, such as `CheckDevice`, are sometimes used live to ensure adherence at run-time (e.g. in device drivers). We are using `Check` purely for static analysis – it is not there at run-time and, therefore, has no impact on performance.

**Safety**

```
VERIFY DEADLOCK.FREE FD
  CheckDevice
```

A CSP failure is a state that a system reaches (represented by its trace to that point) and a set of events with which (if offered by its environment) it may refuse to synchronize. Process `P` failure refines `Q` if (all traces of `P` are traces of `Q`) and (all failures of `P` are failures of `Q`).
Failure refinement makes a powerful statement! \( \varphi \) can only do traces of \( Q \) (so it's safe). More: the failures of \( \varphi \) are allowed by \( Q \). If \( \varphi \) and \( Q \) execute the same trace to a state where their environment offers a set of events that \( Q \) will not refuse, then \( \varphi \) also will not refuse.

A CSP failure is a state that a system reaches (represented by its trace to that point) and a set of events with which (if offered by its environment) it may refuse to synchronise.

Process \( \varphi \) failure refines \( Q \) if (all traces of \( \varphi \) are traces of \( Q \)) and (all failures of \( \varphi \) are failures of \( Q \)).  

We can formalise the expression of the CSP.

Recall our informal understanding of (at least some of) the opening traces of Device (slides 18-35) ...

We can formalise the expression of those traces a bit better ...

What next? ... any order)

Interleave

Recall our informal understanding of (at least some of) the opening traces of Device (slides 18-35) ...

We can formalise the expression of those traces a bit better ...


Recall our informal understanding of (at least some of) the opening traces of Device (slides 18-35) ...

We can formalise the expression of those traces a bit better ...

\[ \langle a, b, a_1, b_1 \rangle, \langle a, a_1, b, b_1 \rangle \]

\[ \langle a, a_1, b, b_1 \rangle, \langle a, b, a_1, b_1 \rangle \]

Interleave

Recall our informal understanding of (at least some of) the opening traces of Device (slides 18-35) ...

We can formalise the expression of those traces a bit better ...

\[ \langle d_0 \rangle || \langle c_1 \rangle || \langle d_0 \rangle \]

\[ \langle d_0 \rangle || \langle c_1 \rangle \]

\[ \langle c_1 \rangle || \langle d_0 \rangle \]

Interleave
We can formalise the expression of those traces a bit better ...

From such trace expressions, we can directly write down an \textit{occam}-\textit{π} process that offers all of them ...

From such trace expressions, we can directly write down an \textit{occam}-\textit{π} process that offers all of them ...

From such trace expressions, we can directly write down an \textit{occam}-\textit{π} process that offers all of them ...

From such trace expressions, we can directly write down an \textit{occam}-\textit{π} process that offers all of them ...

This generation can be automated.

And, still using our intuitive understanding, guess the next cycle of events ...

And, still using our intuitive understanding, guess the next cycle of events ...
Whilst our intuition indicated that the first two lines of DeviceSpec reflected the initial behaviour of Device, it was unclear whether the pattern repeated cleanly as its sub-components started looping.

We also have:

```
VERIFY Device Device REFINES.F DeviceSpec
```

This is all we need. Any traces performed by Device are allowed by DeviceSpec – so it’s safe. Any failures reached by Device are allowed by DeviceSpec – so it’s as alive as DeviceSpec (which was built always to offer everything in the specified trace pattern).

For simplicity, most process arguments are omitted in VERIFY assertions – the `occam-π` compiler supplies all necessary events:

```
VERIFY DEADLOCK.FREE.F Device
```

The CSP channels are names from the `occam-π CHAN` and `BARRIER` parameter names of the asserted process, suffixed by a unique number generated by the compiler.

```
channel a0_42_, b0_42_, c0_42_, a1_42_,
        b1_42_, c1_42_, d0_42_, d1_42_

assert Device (a0_42_, b0_42_, c0_42_, a1_42_,
           b1_42_, c1_42_, d0_42_, d1_42_)
     : [ deadlock free [FD] ]
```
For simplicity, most process arguments are omitted in VERIFY assertions – the compiler supplies all necessary events:

```
VERIFY NOT TERMINATES Device

assert not SKIP [FD=
 Device (a0_42_, b0_42_, c0_42_, a1_42_,
 b1_42_, c1_42_, d0_42_, d1_42_)] \ Events
```

The CSPn channel names are generated from the `occam-n` CHAN and BARRIER parameter names of the asserted process, suffixed by a unique number generated by the compiler.

Subsequent assertions about the same process may reuse channels previously generated. (Note: processes in refinement assertions should have the same parameter signatures, though the formal names can be different).
Reflection

**Reflection**

devices that can be used to provide efficient executables and rich formal analysis.

This presentation reflects a proposal to extend **occam** to include verification assertions (about deadlock, livelock, determinism and refinement). Its compiler will generate suitably abstracted CSP, and interact with the **FD22** model checker, feeding back results in terms of the source **occam** program.

Together with the ancient formal Laws of occam Programming, this moves **occam** towards a process algebra in its own right.

* http://portal.acm.org/citation.cfm?id=53255


---

**A Thesis** (for which we have experimental evidence)

**A Thesis** (for which we have experimental evidence)

Not only

can we (and **should** we) teach concurrency at the start of the undergraduate CS curriculum …

Because it’s there

- Sequence, variables, assignment, parameters, concurrency, channels, synchronisation, …

Fundamental primitives for software engineering

- All are important. All are simple. All are available.

---

**A Thesis** (for which we have experimental evidence)

Not only

can we (and **should** we) teach concurrency at the start of the undergraduate CS curriculum …

Because it’s there

- CSP / π-calculus **occam** / JCS

**Process Orientation**

- for complexity

**Because it simplifies**

- for performance

---

**Reflection**

Observation

Formal verification of the behaviour of concurrent processes can be achieved – by students – engaging in only simple reasoning themselves.

The complexity of synchronisation and communication analysed goes far beyond the embarrassingly parallel.

Aside: model checking found an error overlooked in developing the **Device** case study on paper (the need for ping) … which shows the necessity for formal checks (especially when those responsible think they won’t make mistakes!)

Further reading: Santa Claus: Formal Analysis of a Process Oriented Solution.

* http://doi.acm.org/10.1145/1734206.1734211

TOPLAS, [April, 2010]
A Thesis (for which we have experimental evidence)

Complex and high-performance systems cannot avoid concurrent design, implementation and reasoning.

Common concurrency bugs are intermittent – not repeatable on demand: Untestable in practice.

We stand on the shoulders of giants (who made the theory and model checkers). We verify programs just by writing programs ... it becomes everyday practice.

But also...

we can (and we should) teach formal analysis and verification of this concurrency at the same time ... and it can be done just by ordinary programming (execution code + verification code).

Observation

Can we teach students (those who love to program, anyway) concurrency so that:

- they quickly develop a correct and intuitive understanding of the primitive mechanisms (e.g. processes, communication, synchronisation, networks) and higher level patterns (e.g. client-server, phased barriers, (OC-PAIR)) ...?
- they can use those primitives and patterns with the same fluency as they use serial computing primitives, without tripping over dark hazards ...?
- they can develop their own patterns when the standard ones don't apply ...?
- they can use formal methods to verify good behaviour (e.g. freedom from deadlock and livelock, safety, liveness), without training in the underlying mathematics (process algebra, denotational semantics) ...?
- they can do this as normal everyday practice, without any sense of fear ...?

Postscript ...

accom-π has

a dynamic concurrency model built into its core design ... with full denotational semantics (based on the CSP trace/failure/divergence model)

no data race hazards (eliminated by compiler aliasing analysis)...

deterministic concurrency by default. Non-determinism is introduced only by explicit use of special features (e.g. choice, shared channels)...

the fastest and most effective multithreaded scheduler on the planet (maybe)...

program verification by programming (and a little thinking)...

simple to learn, simple to use (e.g. 90 min Lego Robots occam workshop)...

a tiny user base ... to be fixed (???) ... How (???) when (???) ...

Observation

Can we teach students (those who love to program, anyway) concurrency so that:

- they quickly develop a correct and intuitive understanding of the primitive mechanisms (e.g. processes, communication, synchronisation, networks) and higher level patterns (e.g. client-server, phased barriers, (OC-PAIR)) ...?
- they can use those primitives and patterns with the same fluency as they use serial computing primitives, without tripping over dark hazards ...?
- they can develop their own patterns when the standard ones don't apply ...?
- they can use formal methods to verify good behaviour (e.g. freedom from deadlock and livelock, safety, liveness), without training in the underlying mathematics (process algebra, denotational semantics) ...?
- they can do this as normal everyday practice, without any sense of fear ...?

Postscript ...

accom-π has

a dynamic concurrency model built into its core design ... with full denotational semantics (based on the CSP trace/failure/divergence model)

no data race hazards (eliminated by compiler aliasing analysis)...

deterministic concurrency by default. Non-determinism is introduced only by explicit use of special features (e.g. choice, shared channels)...

the fastest and most effective multithreaded scheduler on the planet (maybe)...

program verification by programming (and a little thinking)...

simple to learn, simple to use (e.g. 90 min Lego Robots occam workshop)...

a tiny user base ... to be fixed (???) ... How (???) when (???) ...

Any questions?