Interface specification & verification

Wim Couwenberg

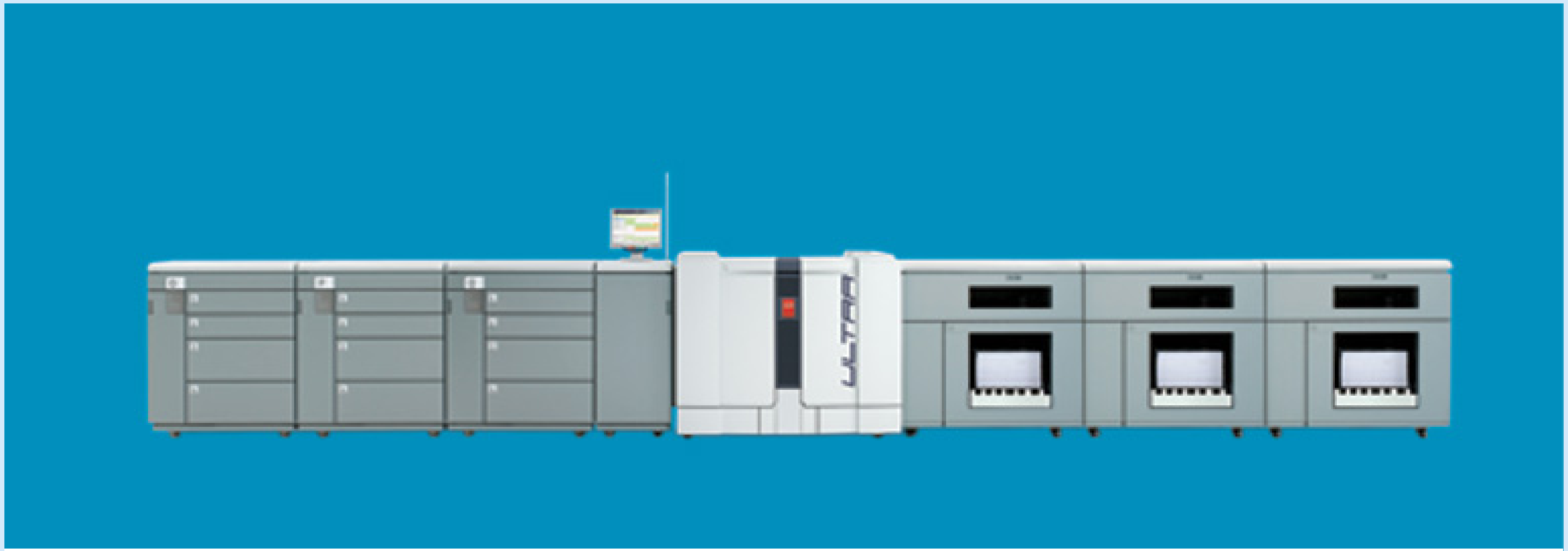
Lua workshop 2011

Overview

- About Océ
- Component technology
- Interface traces
- Trace file analysis
- Specification & verification

About Océ

- Production printing in Color and B/W



About Océ

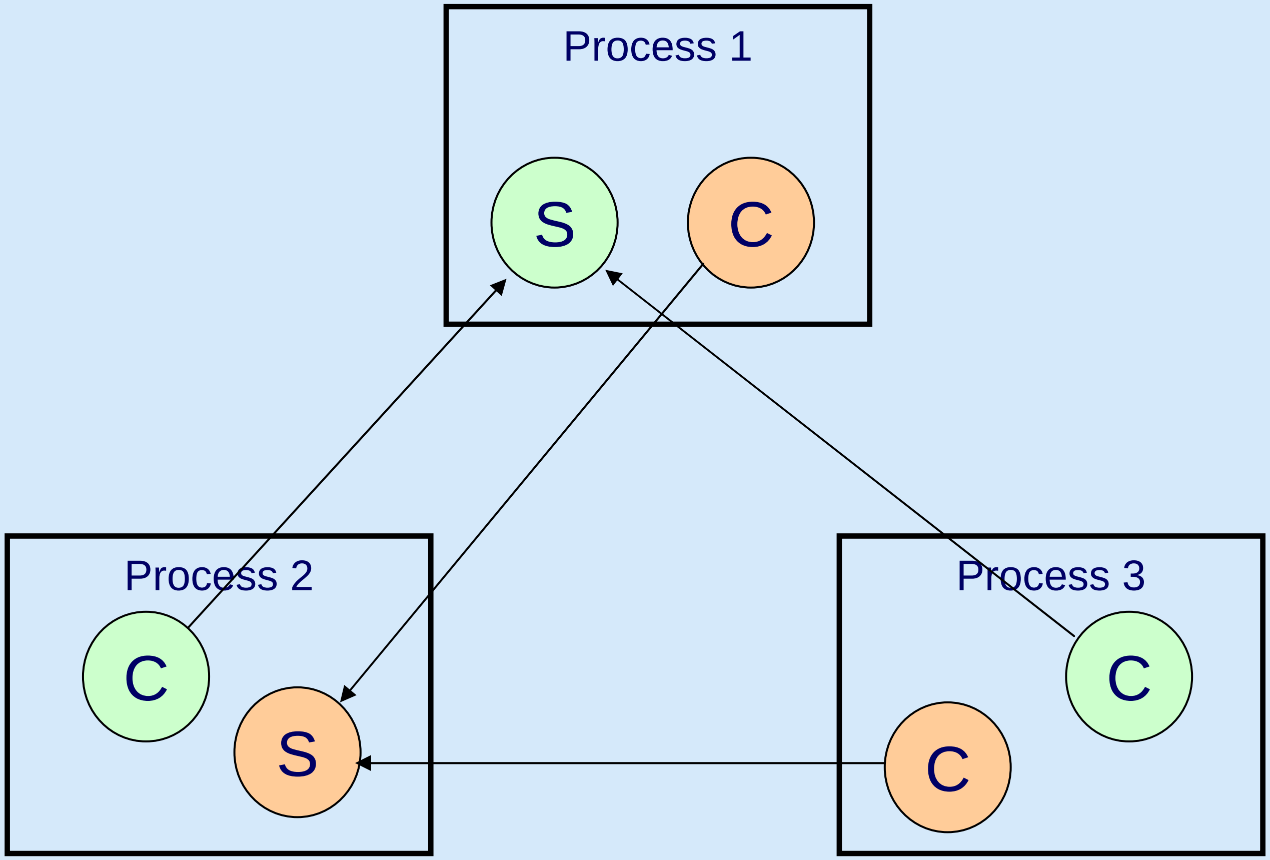
- Founded 1877 in Venlo, The Netherlands
- ~20,000 staff worldwide
- 1,600 specialists at 10 R&D sites in nine countries
- Active throughout printing system value chain
- Worldwide distribution in ~100 countries
- Total revenues 2010: EUR 2.7 billion
- Became a Canon Group company in 2010
- Aiming for global leadership in printing industry

www.oce.com

Component technology

- Single PC distributed software (~50 processes)
- Proprietary component technology
- IDL similar to Corba and COM in XML format
  - basic types
  - structs and arrays
  - interfaces
- Single server multiple clients per IDL file (“protocol”)
  - star topology
  - reference counted object lifetime
- Targets different languages
  - C, C++, C#, Java, Python

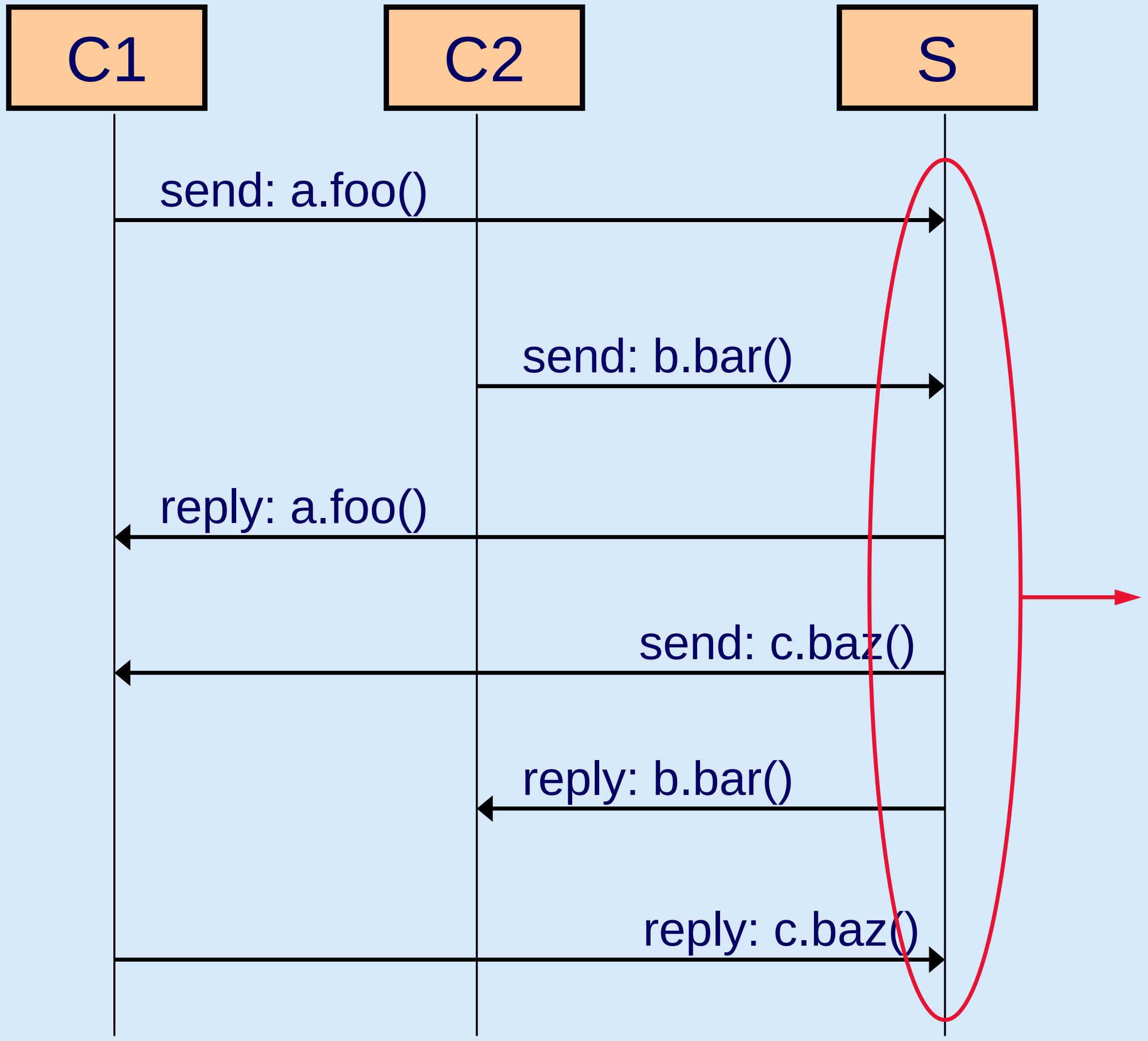
Component technology



Interface traces

- Each method call is a “send” and “reply” pair
  - send: object, method, in parameters
  - reply: out parameters, return value
- Server and clients handle concurrent calls
- Reference count is “just a call”
- Each (binary) send and reply part is logged
  - only server side needed (star topology)
  - timestamp, connection, call id, binary data block
- Logged trace file captures all interactions
  - uniform and complete traces
  - post mortem analysis

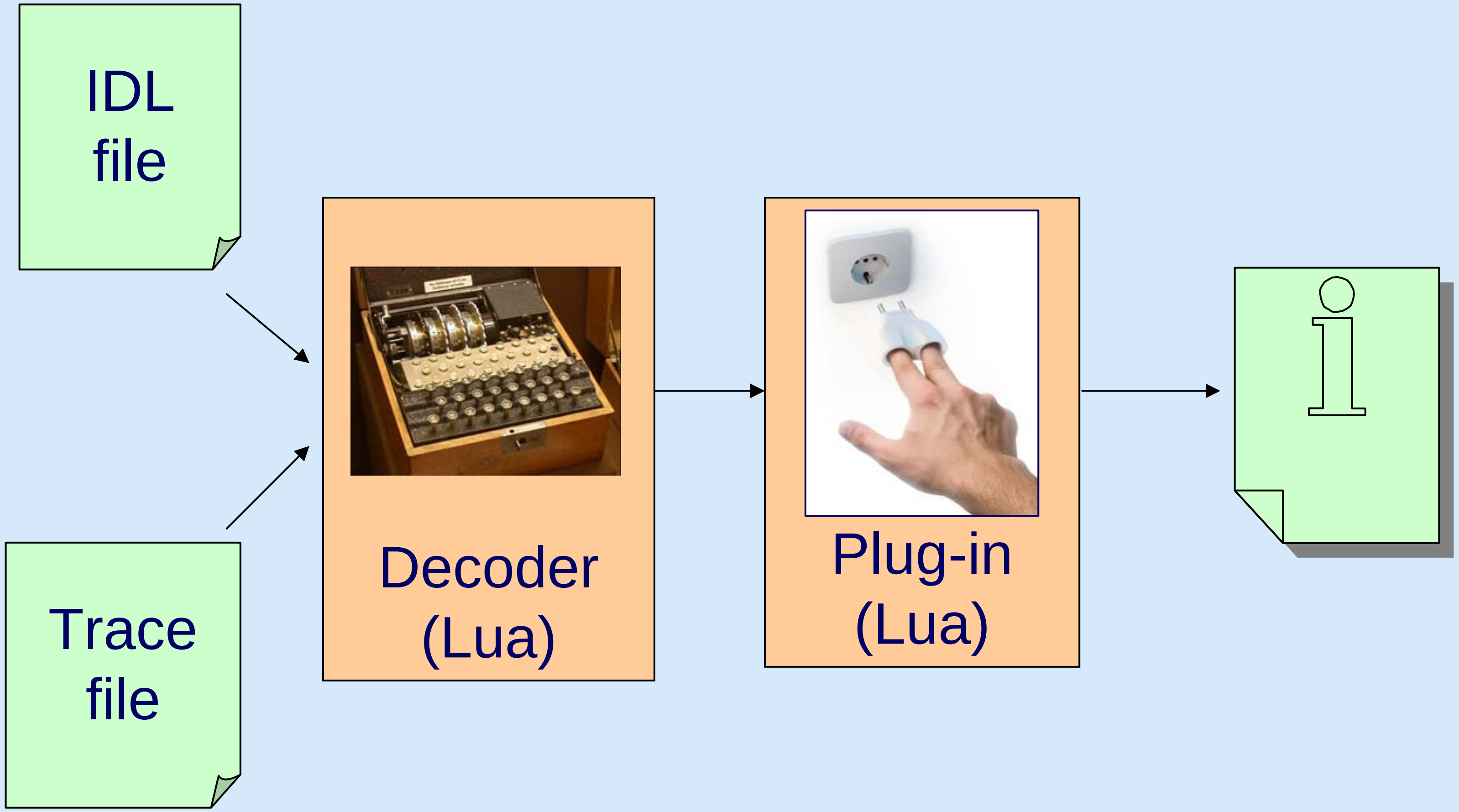
Interface traces



Trace file:

→ 1 send a.foo
→ 2 send b.bar
← 1 reply a.foo
← 3 send c.baz
← 2 reply b.bar
→ 3 reply c.baz

Trace file analysis



Trace file analysis

- Trace file decoder written in 100% Lua
- Parses IDL file
- Reads binary trace file entries (send and reply)
- Demarshals each block and adds meta data
  - timestamp, connection, call id, object, method
  - struct field names, enum symbols
  - in/out and parameters with their names, return value
  - all relevant type information
- Passes each block as a Lua table to a plug-in script
  - generic plug-in for human readable logging
  - generic plug-in to verify protocol compliance
  - specific plug-ins for specific analysis

Specification & verification

- IDL can specify formal requirements in Lua snippets
- Supported specifications
  - constructors for non interface types
  - constructors and destructors for interface types
  - pre and post conditions for interface methods
- Only used for post mortem analysis
- Verification done by a generic decoder plug-in script
  - parses IDL
  - generates verification code from collected snippets
  - checks all send and reply blocks
  - emits warnings or asserts on violations

Specification & verification

- Basic type constructors, typedefs and enums

```
<integer name=“byte”>
  <check>
    -- byte value must be in range
    assert(self >= 0 and self < 256,
      “out of range [value=%d].”, self)
  </check>
</integer>

<enum name=“model”>
  <check>
    warn(self ~= “oldtimer”, “deprecated”)
  </check>
  <item name=“oldtimer”/>
  …
</enum>
```

Specification & verification

- Basic type constructors, struct

```
<struct name=“rectangle”>
  <check>
    -- check top left and bottom right order
    assert(left <= right and top <= bottom,
      “wrong coordinates [%d,%d,%d,%d]”,
      top, left, right, bottom)
  </check>
  <item name=“top” type=“integer”/>
  <item name=“left” type=“integer”/>
  <item name=“bottom” type=“integer”/>
  <item name=“right” type=“integer”/>
</struct>
```

Specification & verification

- Interface constructor & destructor

```
<interface name=“foo”>
  <check type=“ctor”>
    -- setup some state for this object
    self = {
      state = “disconnected”,
    }
  </check>
  <check type=“dtor”>
    -- must be disconnected
    assert(self.state == “disconnected”,
      “wrong state [state=%s]”, self.state)
  </check>
  …
</interface>
```

Specification & verification

- Method pre and post conditions

```
<method name=“connect”>
  <check type=“pre”>
    -- must not be connected yet
    assert(self.state == “disconnected”,
      “wrong state [state=%s]”, self.state)
  </check>
  <check type=“post”>
    -- set state to connected
    self.state = “connected”
  </check>
  …
</method>
```

Specification & verification

- Automatically verify test runs of nightly builds
- Failed assertion also gives context of the failure
- Refers to logging (call id) for further info

```
Failed assertion in foo dtor: wrong state…
[1438] in foo::remove_ref
[0010] foo created in bar::make_foo
[0001] bar created in bar::connect
```

Printing for Professionals