Concurrent Systems, CSP, and FDR

Dyke Stiles & Don Rice

dyke.stiles@ece.usu.edu

http://www.engineering.usu.edu/ece/
Utah State University
June 2001
Why Concurrent Systems Design??

- Many systems are naturally concurrent!!
- Better engineering:
  - Modularity
  - Simplicity
- Reliability & Fault Tolerance
- Speed – on multiple processors
What Are Concurrent Systems?

Any system where tasks run concurrently

- time-sliced on one processor
- and/or on multiple processors
Concurrent Systems

Time-sliced examples:

- Multiple independent jobs
  - Operating system
    - comms, I/O, user management
  - Multiple users’ jobs
- Multithreading within one job
  - C++
  - Java
Concurrent Systems

Multiprocessor examples:

- Distributed memory (message-passing) systems (e.g., Intel, NCube)
- Shared memory systems (e.g., Sun)
Concurrent Systems

Example applications

Numerical computation on multiprocessors

- typically regular communication patterns
- relatively easy to handle
Concurrent Systems

Example applications

Real-time systems on multiple processors

- e.g., flight control, communications routers
- irregular communication, often in closed loops

- difficult to get correct
- may be prone to deadlock and livelock 😞
Concurrent Systems

Example applications

System routines on one multiprocessor node

♦ Manage multiple user tasks

♦ Manage communications
  • Route messages between tasks on node
  • Route messages to tasks on other nodes
  • Manage multiple links to other nodes
  • Manage I/O, interrupts, etc.
Concurrent Systems

Example applications
System routines on one multiprocessor node
Concurrent Systems

Example: complete routing system
What Is “Difficult” About Concurrent Systems?

- Correctness
- Deadlock
- Livelock
Why is Correctness an Issue?

- Multiple processes execute their instructions more or less at the same time.
- The actual operations may interleave in time in a great number of ways:
  - For $n$ processes with $m$ instructions, there are $(nm)!/(m!)^n$ interleavings.
  - Two processes of 10 instructions each have 184,756 interleavings!!
Correctness

Example: the bank balance problem

ATM:

fetch balance
balance = balance − $100
store balance

Payroll Computer:

fetch balance
balance = balance + $1000
store balance
Bank Balance
Original balance = $1000

Interleaving 1:

<table>
<thead>
<tr>
<th>ATM</th>
<th>Payroll Computer</th>
</tr>
</thead>
<tbody>
<tr>
<td>$t_1$</td>
<td>fetch $1000</td>
</tr>
<tr>
<td>$t_2$</td>
<td>balance = $1000 - $100</td>
</tr>
<tr>
<td>$t_3$</td>
<td>store $900</td>
</tr>
<tr>
<td>$t_4$</td>
<td>fetch $900</td>
</tr>
<tr>
<td>$t_5$</td>
<td>balance = $900 + $1000</td>
</tr>
<tr>
<td>$t_6$</td>
<td>store $1900</td>
</tr>
</tbody>
</table>

Final balance = $1900: Correct!
Bank Balance

Original balance = $1000

Interleaving 2:

<table>
<thead>
<tr>
<th>Time</th>
<th>ATM</th>
<th>Payroll Computer</th>
</tr>
</thead>
<tbody>
<tr>
<td>t₁</td>
<td>fetch $1000</td>
<td>fetch $1000</td>
</tr>
<tr>
<td>t₂</td>
<td></td>
<td>balance = $1000 + $1000</td>
</tr>
<tr>
<td>t₃</td>
<td></td>
<td>store $2000</td>
</tr>
<tr>
<td>t₄</td>
<td>balance = $1000 - $100</td>
<td></td>
</tr>
<tr>
<td>t₅</td>
<td>store $900</td>
<td></td>
</tr>
</tbody>
</table>

Final balance = $900: WRONG!
Bank Balance

Only 2 of the twenty possible interleavings are correct!!

Concurrent systems must have some means of guaranteeing that operations in different processes are executed in the proper order.
Deadlock

All processes stopped:

- often because each is waiting for an action of another process
- processes cannot proceed until action occurs
Deadlock

Example: Shared Resource

Two processes wish to print disk files. Neither can proceed until it controls both the printer and the disk; one requests the disk first, the other the printer first:

<table>
<thead>
<tr>
<th>Proc A</th>
<th>Proc B</th>
</tr>
</thead>
<tbody>
<tr>
<td>t1 acquire disk</td>
<td></td>
</tr>
<tr>
<td>t2</td>
<td>acquire printer</td>
</tr>
<tr>
<td>t3 try to acquire printer</td>
<td>DEADLOCK!!</td>
</tr>
</tbody>
</table>
Livelock

- Program performs an infinite unbroken sequence of internal actions.
- Refuses (unable) to interact with its environment.
- Outward appearance is similar to deadlock - but the internal causes differ significantly.
- Example: two processes get stuck sending error messages to each other.
Concurrent Designs Requires:

- Means to guarantee correct ordering of operations
- Models to avoid and tools to detect
  - Deadlock
  - Livelock
CSP: A Solution

Communicating Sequential Processes (CSP)

- Processes interact only via explicit blocking events.
  - Blocking: neither process proceeds until both processes have reached the event.
- There is absolutely no use of shared variables outside of events.
- Can be done - with care – from semaphores, wait, etc.
CSP

A process algebra –

Provides formal (mathematical) means and CASE tools for

- Describing systems of interacting concurrent processes
- Proving properties of concurrent systems
  - Agreement with specifications
  - Deadlock freedom
  - Divergence freedom
CSP Design Philosophy

- Complex applications are generally far easier to design as systems of
  - many small, simple processes
  - that interact only via explicit events.
- Unconstrained use of shared memory can lead to designs that
  - are extremely difficult to implement
  - are not verifiable
CSP Design Example

Virtual Channel System

- Two processes must be able to send identifiable messages over a single wire.
- Solution: append channel identifier to messages, and wait for ack to control flow.
CSP Design Example

Router: single process design
- Software state machine
- State variables are the message states:
  - 0: waiting to input
  - 1: waiting to send downstream
  - 2: waiting for ack
- Result: $3 \times 3 = 9$ state case statement
CSP Design Example

Router: single process design

Example case clause:

\[(S0 = \text{input0}, S1 = \text{input1})::
\]
\[
\text{Read(channel0, channel1)}
\]
\[
\quad \text{If (channel0)}
\]
\[
\quad \quad \text{write data.0}
\]
\[
\quad S0 = \text{send0};
\]
\[
\quad \text{Else}
\]
\[
\quad \quad \text{write data.1}
\]
\[
S1 = \text{send1};
\]
CSP Design Example

Router: single process design

- Nine states – not too bad, but complex enough to require care in the implementation.
- But: if we add another input, it goes to 27 states, and a fourth gives us 81 states!!!
- What are your odds of getting this right the first time?
- Would debugging 81 states be much fun???
CSP Design Example

Router: multiple process design

- One process to monitor each input and wait for the ack (these are identical)
- One multiplexer process to send the inputs downstream
- One demultiplexer process to accept and distribute the acks
CSP Design Example

Router: multiple process design: block diagram
CSP Design Example

Router: multiple process design

Input process:

While (true)

read input;

write input to Mux;

wait for ack from DeMux;
CSP Design Example

Router: multiple process design

Mux process

While (true)

read (input0, input1)

if (input0) write data.0

else write data.1;
CSP Design Example

Router: multiple process design

DeMux process

While (true)

read ack;

if (ack == 0) write ack0
else write ack1;
CSP Design Example

- Router: multiple process design; Summary
  - Three processes – 4 lines each!!
  - Add another input?
    - Add one input process
    - Mux modified to look at 3 inputs
    - Demux modified to handle 3 different acks
- Which implementation would you rather build?
Formal Methods

- Formal methods: mathematical means for designing and proving properties of systems.
- Such techniques have been in use for decades in
  - Analog electronics
    - Filter design: passband, roll-off, etc
    - Controls: response time, phase characteristics
Formal Methods

Digital design

− Logic minimization
− Logical description to gate design
− Formal language description of algorithm to VLSI masks
  (e.g., floating-point processor design)
Formal Methods

Two methods of formal design:

1. **Derive** a design from the specifications.
2. **Assume** a design and prove that it meets the specifications.
CSP

- CSP: deals only with interactions between processes.
- CSP: does not deal (easily) with the internal behavior of processes.
- Hence other software engineering techniques must be used to develop & verify the internal workings of processes.
CSP

The two components of CSP systems:

- *Processes*: indicated by upper-case: \( P, Q, R, \ldots \)
- *Events*: indicated by lower-case: \( a, b, c, \ldots \)
CSP

Example: a process $P$ engages in events $a$, $b$, $c$, $a$, and then $STOP$s:

$$P = a \rightarrow b \rightarrow c \rightarrow a \rightarrow STOP$$

“$\rightarrow$” is the prefix operator;

$STOP$ is a special process that never engages in any event.
CSP Example

A practical example: a simple pop machine accepts a coin, returns a can of pop, and then repeats:

- \( PM = \text{coin} \rightarrow \text{pop} \rightarrow PM \)
- Note the recursive definition - which is acceptable; substituting the \( rhs \) for the occurrence of \( PM \) in the \( rhs \), we get
  - \( PM = \text{coin} \rightarrow \text{pop} \rightarrow \text{coin} \rightarrow \text{pop} \rightarrow PM \)
- (RT processes are often non-terminating.)
CSP Example

The router:

- **ch0** → **In0** → **toMux0** → **Mux** → **down**
- **ch1** → **In1** → **toMux1** → **DeMux** → **up**
  - **ack0** from **Mux** to **In0**
  - **ack1** from **DeMux** to **In1**
The router processes: Input

\[ \text{In0} = \text{ch0?}x \rightarrow \text{toMux0!}x \rightarrow \text{ack0} \rightarrow \text{In0} \]
The router processes: Mux

\[ \text{Mux} = \begin{cases} \text{toMux0}\,?x \rightarrow \text{down}!x.0 \rightarrow \text{Mux} \\ \neq \text{toMux1}\,?x \rightarrow \text{down}!x.1 \rightarrow \text{Mux} \end{cases} \]
The router processes: DeMux

DeMux = up?x →
    (ack0 ⇐ x == 0 ⇒ ack1)
    → DeMux
CSP

Example: the process graph of a data acquisition system (NB: no arrows...):

- DataSampler
  - get_data
  - data_ready
- DataAq
  - send_data
- DataStore
CSP

- DataAq: waits until it is notified by the sampler that data is ready, then gets and transforms the data, sends it on to be stored, and repeats:
  \[ \text{DataAq} = \text{data\_ready} \rightarrow \text{get\_data} \rightarrow \text{send\_data} \rightarrow \text{DataAq} \]

- Note that the transform is an internal process and is not visible; \text{data\_ready}, \text{get\_data}, and \text{send\_data} are events engaged in with other processes.
CSP

- The data sampling process would engage in the events *data_ready* and *get_data*:

  \[ \text{DataSampler} = \text{data\_ready} \rightarrow \text{get\_data} \rightarrow \text{DataSampler} \]

- Data store engages only in *send_data*:

  \[ \text{DataStore} = \text{send\_data} \rightarrow \text{DataStore} \]
CSP

- We thus have three processes, each of which has an alphabet of events in which it can engage:
  - *DataSampler*: $AS_a = \{\text{data\_ready, get\_data}\}$
  - *DataAq*: $ADA = \{\text{data\_ready, get\_data, send\_data}\}$
  - *DataStore*: $AST = \{\text{send\_data}\}$
- The entire alphabet of the composite process is denoted by $\Sigma$. 
CSP

- The entire data acquisition system would be indicated by the *alphabetized parallel* composition of the three processes:

\[
DAS = \text{DataSample}_{ASa} \parallel \text{DataAq}_{ADA} \parallel \text{DataStore}_{AST}
\]

- Two processes running in alphabetized parallel with each other must agree (synchronize) on events which are common to their alphabets.
CSP Details

Traces

- The *traces* of a process is the set of all possible sequences of events in which it can engage.
- The traces of *Data_Store* are simple:
  - \{<>, <send_data>^n, 0 \leq n \leq \infty\}
  - <> is the empty trace.
CSP Details

Traces

DataAq can have engaged in no events, or any combination of the events data_ready, get_data, and send_data in the proper order:
CSP Details

Traces of DataAq:

\[ \text{traces(DataAq)} = \{ <>, <\text{data\_ready}>, <\text{data\_ready, get\_data}>, <\text{data\_ready, get\_data, send\_data}>^n, <\text{data\_ready, get\_data, send\_data}>^n \land <\text{data\_ready}>, <\text{data\_ready, get\_data, send\_data}>^n \land <\text{data\_ready, get\_data}>, 0 \leq n \leq \infty \} \]
CSP Details

- Traces specify formally what a process can do - if it does anything at all.
- This is a safety property: the trace specification should not allow any unacceptable operations (e.g., we would not want to allow two stores without an intervening new sample; thus \(<...send\_data, send\_data...>\) is ruled out.
CSP Details

- Traces do not force a process to do anything.
- We force action by limiting what a process can refuse to do. This is a *liveness* property.
CSP Details

- **refusal set**: a set of events which a process can refuse to engage in regardless of how long they are offered.

- E.g., the refusal set of \(DataAq\) after it has engaged in \(data\_ready\) is \{\(data\_ready, send\_data\)\}. 
CSP Details

Refusals can be shown nicely on the transition diagram of *DataAq*:

\[
\{\text{data\_ready, send\_data}\}
\]

\[
\text{data\_ready} \quad \text{get\_data}
\]

\[
\{\text{get\_data, send\_data}\}
\]

\[
\{\text{data\_ready, get\_data}\}
\]
CSP Details

- A *failure* is a pair \((s, X)\), where \(s\) is a trace and \(X\) is the set of events which are refused after that trace.
- We force a process to do the right things by specifying the acceptable failures - thus limiting the failures it can exhibit.
CSP Details

Failures

E.g., $DataAq$ cannot fail to accept a new $data\_ready$ event after a complete cycle; its failures cannot contain $<data\_ready, get\_data, send\_data>^n, \{data\_ready\})$. 
CSP Details

- **traces:**
  - specify what *can* be done
- **failures:**
  - specify allowed failures
- Together, these guarantee that the appropriate things *will* be done.
- We have only to prevent deadlock and livelock...
CSP Details

Deadlock freedom:

A system is deadlock free if, after any possible trace, it cannot refuse the entire alphabet $\Sigma$:

$$\forall s . (s, \Sigma) \notin failures(DAS)$$
CSP Details

Livelock (divergence) freedom:

- *divergences* of a process:
  the set of traces after which the process can enter an unending series of internal actions.

- A system is divergence free if there are no traces after which it can diverge:
  \[
  \text{divergences}(DAS) = \{\}
  \]
CSP Details

- A complete specification:
  - Acceptable traces
  - Acceptable failures
  - Deadlock freedom
  - Divergence freedom

- These properties can be checked by rigorous CASE tools – from FSE Ltd.
CSP Details

Refinement

- A specification is often a process that exhibits all acceptable implementations - which may be overkill, but easy to state.
- Implementation $Q$ refines specification $P$ ($P \subseteq Q$) if:
  
  $Q$ satisfies the properties of $P$:
  - the traces of $Q$ are included in the traces of $P$;
  - the failures of $Q$ are included in the failures of $P$. 
CSP Details

Refinement of a design problem:

- Initial specification:
  - very general (often highly parallel)
  - correctness easy to verify.

- CASE tools:
  verify that a particular implementation (whose correctness may not be obvious) properly refines the original specification.
CSP Details

Algebraic manipulations

- Objects and operations within CSP form a rigorous algebra.
- Algebraic manipulations:
  - demonstrate the equivalence of processes
  - transform processes into ones that may be implemented more efficiently.
CSP Details

Algebraic manipulations: simple laws

- Alphabetized parallel composition obeys commutative laws
  \[ P_{A \parallel B} Q = Q_{B \parallel A} P \]
  and associative laws
  \[ (P_{A \parallel B} Q)_{B \parallel C} R = P_{A \parallel B} (Q_{B \parallel C} R) \]
- and many, many more...
CSP Details

Algebraic manipulations: step laws

*Step laws:*

convert parallel implementations into equivalent sequential (single-thread) implementations:
CSP Details

Step law example:

Assume $P = ?x:A \rightarrow P'$ and $Q = y:B \rightarrow Q'$

$P \parallel_{B} Q = ?x:(A \cup B) \rightarrow P \parallel_{B} Q'$

$\preceq x \in (A \cap B) \npreceq

$P \parallel_{B} Q$

$\preceq x \in A \npreceq

$P \parallel_{B} Q'$

Repeated application results in a sequence of events.
CSP Details

Sequentialization

◆ The parallel composition of the DataAq and DataStore can be sequentialized - which may be more efficient on a single processor:

\[ DataAq_{ADA} \parallel_{AST} DataStore = DaDst = \]
\[
data_{\text{ready}} \rightarrow \text{get}_{\text{data}} \rightarrow \text{send}_{\text{data}} \rightarrow DaDst
\]

◆ The CASE tools will verify that the sequential version refines the concurrent version.
CSP Tools

ProBE

Process Behaviour Explorer

- Allows manual stepping through a CSP description
- Shows events acceptable at each state
- Records traces
- Allows manual check against specifications
CSP Tools

FDR (a model checker)

Failures-Divergences-Refinement

Mathematically tests for:

• Refinement of one process against another
  – Traces
  – Failures
  – Divergences
• Deadlock freedom
• Divergence freedom
CSP Compatibility

- “My work group uses the (Yourdon, Booch, UML, PowerBuilder, Delphi… software development system); can I still use CSP?”

- Certainly – CSP can be used wherever you design with processes that interact only via CSP-style explicit events.
CSP Compatibility

“CSP seems to be based on message passing; Can I use it with locks, critical sections, semaphores, mutexes and/or monitors???”

Absolutely! As long as your processes interact only via explicit locks, mutexes, etc., CSP can describe them – and prove them.
CSP Mutex

Modeling of shared-memory primitives

**Mutex:**

```c
claim mutex1;
modify shared variable;
release mutex1;
```
CSP Mutex

A CSP mutex process:

\[ \text{Mutex1} = \text{claim} \rightarrow \text{release} \rightarrow \text{Mutex1} \]

The process will not allow a second claim until a prior claim has been followed by a release.
CSP Mutex

Weaknesses:

- Compiler does not require use of mutex to access shared variables.
- A process may neglect to release the mutex, thus holding up further (proper) accesses.
CSP Mutex

A more robust version that allows only the process making the claim to complete the release:

\[
\text{RMutex} = \begin{align*}
\text{claim?ProcID} & \rightarrow \text{release!ProcID} \\
& \rightarrow \text{Rmutex}
\end{align*}
\]
CSP Mutex

Use of the robust mutex:

Proc 29:
   claim!29;
   modify shared variable;
   release?29;
CSP Mutex

The way it should be done: the shared variable is modifiable only by a single process (which allows a read as well):

\[
\text{Robust}(x) = \\
\text{ModifyX!y} \rightarrow \text{Robust}(x + y) \\
\not\exists \text{readX?x} \rightarrow \text{Robust}(x)
\]
Semaphores

Definitions

\( \langle x; \rangle: \text{operation } x \text{ is atomic} \)

Claim semaphore \( s \):

\[ \text{P}(s): \langle \text{await } (s > 0) \ s = s - 1; \rangle \]

Release semaphore \( s \):

\[ \text{V}(s): \langle s = s + 1; \rangle \]
Semaphores

A semaphore process (initialized to \( s = 1 \)):

\[
\begin{align*}
\text{SemA} &= \text{SemA1}(1) \\
\text{SemA1}(s) &= \begin{cases} 
\text{(pA} \rightarrow \text{SemA1}(s-1)) \wedge s > 0 \wedge \text{STOP} 
\end{cases} \\
\ne
\end{align*}
\]
Summary 1

Thirty+ years of experience shows that
- Complex applications are generally far easier to design as systems of
  - many (2 – 2000) small, simple processes
  - that interact only via explicit events.
- Careless use of shared memory can lead to designs that
  - are extremely difficult to implement
  - are not verifiable
  - are wrong!
Summary 2

CSP + Tools:

- Clean, simple specification of concurrent systems
- Rigorous verification against specifications
- Proof of deadlock and livelock freedom
- Verifiable conversion between concurrent and single-threaded implementations
- Works with any process-oriented development system.
CSP Applications

- Real-time & embedded systems
- Communications management
- Communications security protocols
- Digital design – from gate-level through FPGAs to multiple systems on a chip
- Parallel numerical applications
- Algorithm development
Example:
Ring Network Router

Don Rice, Bin Cai, Pichitpong Soontornpipit
ECE 6750 Class Project
http://www.engineering.usu.edu/ece/
Utah State University
Ring Network Description

■ Three nodes connected in a ring topology
■ Two inputs and two outputs per node
■ One transmit/receive pair between nodes
■ Input must be acknowledged by destination before additional input is accepted
■ Error-free network: packets are not lost, damaged, or duplicated
Three Two-Input Node Ring

This configuration sometimes deadlocks!

July 5, 2001

Copyright G. S. Stiles 2001
Design Procedure

- Began with two-node topologies in CSP
- Used ProBE and FDR to explore designs
  - Identified deadlock scenarios
  - Verified deadlock-free design
- Implemented application with Java CTJ
- Ported to JCSP applet
Two-Input/Two-Output Node

- Inputs `upin0`, `upin1` accept data value and destination ID [0,5]
- Outputs `downout0`, `downout1` produce data value, source ID [0,5]
- Data flows on solid lines (e.g., `uptodown` bus,) acknowledgments flow on dashed lines (e.g., `downtoup` bus)
Input Handler: “Upstream”

(from output handler)

upin0

UpHandler10

UpHandler11

Mux1

UpCntrl1

downtoup1

uptodown1

(data1_1)

(ack1_1)
Sample Code from “UpHandler”

Java processes developed from CSP are typically very simple.

```java
public void run()
{
    intArray packet = null; // packet from test source class
    ChanIO UpH = new ChanIO("UpHandler"+Identity); // IO wrapper
    int ack = 0; // acknowledgment from destination
    boolean Running = true; // allow for external control someday

    // Repeatedly read data and pass it on:
    while(Running)
    {
        packet = UpH.Read(input, " d.d" ); // Read destination, data from test source
        UpH.Write(output, packet, " d.d" ); // Write destination, data to Mux
        ack = UpH.Read(ackin, " ack" ); // Wait for ack from UpCntrl
    }
    // End while.

    // End run
}
```

Read() and Write() methods were wrappers for CTJ try/catch clauses; wrappers were converted to JCSP with little impact on router functions.

July 5, 2001

Copyright G. S. Stiles 2001
Output Handler: “Downstream”

DeMux1

(from previous node)

downtoup0
(implicit multiplexing using Any2OneChannel)

Downtoup0

DownHandler10

downout0

downout1

DownHandler11

DownMux1Bus

Data1_1

(to input handler)

Ack1_1

Uptodown0
Deadlock Prevention

- Used FDR to evaluate alternatives
- A single buffer added to the system was necessary and sufficient to prevent deadlock
- Added “passthru” buffer to Node 1 in Java version
Conclusions

- Design in CSP with FDR testing and verification provides confidence not possible with Java trial-and-error testing
- Model optimization was critical to operate FDR in student lab environment
- Conversion from CSP to Java CTJ or JCSP is largely cut-and-paste exercise once basic examples are provided… (designers had little prior Java experience)
Related USU Projects

- Creation of Java code directly from CSP
  E.g., the simple router
- Automatic conversion of CSP from parallel to sequential
- Compilation of Java to VHDL/FPGA
- Analysis of autonomous vehicle software
- Analysis of internet protocols
Courses:

- **ECE 5740**
  - Concurrent Programming (under Win32)
  - Fall
- **ECE 6750**
  - Concurrent Systems Engineering I (CSP I; Java)
  - Spring
- **ECE 7710**
  - Concurrent Systems Engineering II (CSP II; Java, C)
  - Add real-time specifications
  - Alternate Falls