

# Design, Verification, and Testing of Synchronization and Communication Protocols with Java

G. S. Stiles, D. D. Rice,  
and J. R. Doupnik  
Electrical and Computer Engineering  
Utah State University

5 July 2001

Copyright G. S. Stiles 2001

1

## Introduction

Communication and Synchronization –  
an important part of the curriculum:

- Networking – all levels!
- Distance Education Systems
- Real-time & Embedded Systems
- Concurrent Systems Design
  - Operating Systems
  - Applications

5 July 2001

Copyright G. S. Stiles 2001

2

## Introduction

- Concurrent design: an important part of software engineering:
  - Modular design, with
  - small, simple modules...
  - that run concurrently, and
  - interact infrequently.
- Much easier than a single, large program!!

7/5/01

Copyright G. S. Stiles 2001

3

## Introduction

### Possible platforms:

- Visual C++
  - Complex concurrency features
  - A year or more of experience
  - .. and some OS experience
- Java
  - Simple concurrency model
  - Widely taught at the introductory level

7/5/01

Copyright G. S. Stiles 2001

4

# Java Concurrency

## Concurrency support:

- Simple thread model
- Mutual exclusion via *synchronized*:
  - Objects
  - Methods
- A limited conditional wait
- Shared variables
- Message-passing libraries
- Many texts

7/5/01

Copyright G. S. Stiles 2001

5

# Java Concurrency

## The Java *synchronized* primitive

- Each object has a hidden lock controlling access to code marked as *synchronized*.
- Only one thread at a time may execute a *synchronized* block of code.

7/5/01

Copyright G. S. Stiles 2001

6

# Java Concurrency

## Conditional Wait

- If a condition is not satisfied, *wait()* can be called – releasing the lock.
- *notify* (or *notifyAll*) wakes the waiting threads.

7/5/01

Copyright G. S. Stiles 2001

7

# Java Concurrency

- Caution!
  - Java does not require that access to shared resources be synchronized.
  - The Java specification does not say which thread is awakened on a *notify*.
- These operations must be used very carefully!

7/5/01

Copyright G. S. Stiles 2001

8

# CSP

- CSP: a process algebra for dealing with interactions between processes.
- The notation is simple and intuitive.
- CSP does not deal (easily) with the internal behavior of processes.

7/5/01

Copyright G. S. Stiles 2001

9

# CSP

The two components of CSP systems:

- *Processes*: indicated by upper-case: P, Q, R, ...
- *Events*: indicated by lower-case: a, b, c, ...

7/5/01

Copyright G. S. Stiles 2001

10

# CSP

Example: a process  $P$  engages in events  $b$ ,  $c$ ,  $a$ , and then refuses any further action:

$$P = b \rightarrow c \rightarrow a \rightarrow \text{STOP}$$

“ $\rightarrow$ ” is the *prefix* operator; STOP is a special process that never engages in any event.

7/5/01

Copyright G. S. Stiles 2001

11

# CSP

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

$$\text{PM} = \text{coin} \rightarrow \text{pop} \rightarrow \text{PM}$$

7/5/01

Copyright G. S. Stiles 2001

12

# CSP

A customer who purchases only one can,  
consumes it, and then terminates:

`Cust = coin → pop → drink → STOP`

7/5/01

Copyright G. S. Stiles 2001

13

# CSP

The pop machine and the customer  
run in parallel:

`System = PM [ | A | ] Cust`

and synchronize on the alphabet

`A = {coin, pop}`

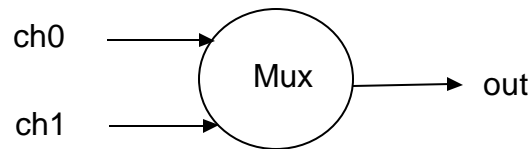
7/5/01

Copyright G. S. Stiles 2001

14

## CSP

A multiplexer that accepts an input from either channel 0 or channel 1, passes it out over the channel out, and then repeats:



```

Mux =  ch0?x → out!x → Mux
      [ ]
      ch1?x → out!x → Mux
  
```

7/5/01

Copyright G. S. Stiles 2001

15

## CSP and Java Design Procedure

- Design in CSP
- Verify the CSP with the FDR CASE tools:
  - Correctness
  - Deadlock
  - Livelock
- Implement and test in Java

7/5/01

Copyright G. S. Stiles 2001

16



## Shared Memory Synchronization – the bank balance problem

Original balance = \$1000

Interleaving 1:

	<u>ATM</u>	<u>Payroll Computer</u>
t1	fetch \$1000	
t2	balance = \$1000 - \$100	
t3	store \$900	
t4		fetch \$900
t5		balance = \$900 + \$1000
t6		store \$1900

Final balance = \$1900: Correct!

7/5/01

Copyright G. S. Stiles 2001

17

## The bank balance problem

Original balance = \$1000

Interleaving 2:

	<u>ATM</u>	<u>Payroll Computer</u>
t1	fetch \$1000	
t2		fetch \$1000
t3		balance = \$1000 + \$1000
t4		store \$2000
t5	balance = \$1000 - \$100	
t6	store \$900	

Final balance = \$900: WRONG!

7/5/01

Copyright G. S. Stiles 2001

18

## Bank Balance: Java

Solution:

force the fetch-store-update  
sequence to be executed  
atomically.

In Java: use a synchronized method (which returns the  
new balance):

```
public synchronized
    float update_balance(float deposit);
```

7/5/01

Copyright G. S. Stiles 2001

19

## Bank Balance: Modeling in CSP

Create a CSP process that will  
synchronize with all customers and  
force the update to be done atomically.

First the customer:

```
Customer =    enter!deposit ->
              exit?new_balance ->
              Customer
```

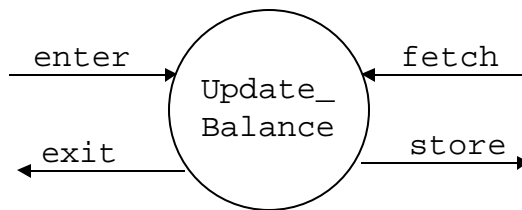
7/5/01

Copyright G. S. Stiles 2001

20

## Bank Balance: CSP

The synchronization process:  
 accept enter request from the customer  
 fetch old balance  
 store new balance  
 return new balance to customer



7/5/01

Copyright G. S. Stiles 2001

21

## Bank Balance: CSP

The synchronization process:

```

Update_Balance =
    enter?deposit ->
    fetch?balance ->
    store!(balance + deposit) ->
    exit!(balance + deposit)->
    Update_Balance
  
```

7/5/01

Copyright G. S. Stiles 2001

22

## Bank Balance: CSP

Multiple customers interleave –  
and do not interact with each other:

```
Customers =
    Customer1 |||
    Customer2 |||
    ... ||| CustomerN
```

7/5/01

Copyright G. S. Stiles 2001

23

## Bank Balance: CSP

The complete system consists of  
the customers running in parallel with  
the update process and synchronizing  
on the enter and exit events:

```
System =    Customers
           [| A |]
           Update_Balance
where A = {enter, exit}
```

7/5/01

Copyright G. S. Stiles 2001

24

## Bank Balance: Check the CSP

Correct operation: only one customer  
is in the critical update section at a time;  
enforce by requiring the enter and exit events  
to alternate:

```
Safety_Spec =      enter.x ->
                   exit.y ->
                   Safety_Spec
```

7/5/01

Copyright G. S. Stiles 2001

25

## Bank Balance: Check the CSP

The CSP CASE tool FDR will verify that  
all possible behaviors of the System  
satisfy the safety specification.

7/5/01

Copyright G. S. Stiles 2001

26

## Bank Balance: CSP

A more robust version:

add a customer ID and require  
that successive `enters` and `exits`  
have the same ID.

7/5/01

Copyright G. S. Stiles 2001

27

## Message Passing

CSP-style message-passing libraries for Java:

- JCSP (University of Kent at Canterbury)
- CTJ (University of Twente)

... available on the web:

- <http://www.cs.ukc.ac.uk/projects/ofa/jcsp/>
- <http://www.rt.el.utwente.nl/javapp/>

7/5/01

Copyright G. S. Stiles 2001

28

## Nagle Mode Enhancement

TCP messages:

- Messages broken into packets for transmission
- Each packet requires ACK
- Save bandwidth via Nagle mode: ACK only after every second or third packet – or timeout (0.2 s)

7/5/01

Copyright G. S. Stiles 2001

29

## Nagle Mode Enhancement

TCP messages

- But: if message is not a multiple of the packet size, we have a “small tail” at the end;
- – a waste of bandwidth, so hold until another message arrives or timeout.
- This may result in a significant delay!
- Short messages: max 5 per second!

7/5/01

Copyright G. S. Stiles 2001

30

## Nagle Mode Enhancement

The Douppnik solution:

- Transmit small tail immediately if it is the last of the application's data;
- otherwise hold the tail for arrival of more application data.
- Result: significant improvement in performance!!

7/5/01

Copyright G. S. Stiles 2001

31

## Nagle Mode Enhancement

- The problem:
  - Verify improvement with CSP
- The approach:
  - Assume a clock that produces regular *tocks*.
  - Nagle mode will not be able to transmit a short tail until a timeout (a tock) occurs
  - Enhanced mode will transmit the short tail prior to the tock.

7/5/01

Copyright G. S. Stiles 2001

32



## Nagle Mode Enhancement

- Assume 1 packet = 2 “chunks”
- A 3-chunk message: 1 packet plus a short tail
- A transmission of 2 chunks (one packet):  
    send! 2

7/5/01

Copyright G. S. Stiles 2001

33

## Nagle Mode Enhancement

- The original Nagle mode will not transmit the third chunk until the 200 ms timeout (a tock) occurs.
- Thus the original Nagle mode cannot transmit the message with no intervening tocks.

7/5/01

Copyright G. S. Stiles 2001

34

## Nagle Mode Enhancement

The specification:

Under the enhanced mode, a message with a short packet must be able to be transmitted with no intervening tocks:

```
TCP_SPEC =
  start -> send?2 -> send?1 ->
  finish -> STOP
```

7/5/01

Copyright G. S. Stiles 2001

35

## Nagle Mode Enhancement

Verification with FDR:

- FDR verifies that the original Nagle mode cannot meet the spec.
- FDR verifies that the enhanced Nagle mode can transmit the 3-chunk message with no intervening tocks.

7/5/01

Copyright G. S. Stiles 2001

36

## Conclusions

- CSP provides an intuitive method for describing synchronization and communication protocols.
- FDR supplies the tools to verify the correctness of the protocols.
- Java + CSP libraries provides the means for implementing and testing the protocols.

7/5/01

Copyright G. S. Stiles 2001

37

## The fast track to success:

- Design with CSP
- Verify with FDR
- Implement in Java with little pain!
- Students readily handle systems with up to 60 or so concurrent processes.

7/5/01

Copyright G. S. Stiles 2001

38