# Assignment 2

EECE 7368

Aly Sultan 001057787

Professor Gunnar Schirner

# Table of contents



## Problem 2.1: Language Concepts

We have discussed the importance of separation of concerns.

(a) What is separation of concerns?
It is the concept of separation of different components of a program into distinct sections where each section addresses a separate concern.

(b) Name two concerns that should be separated in a language as they cover orthogonal aspects?
Separation of computation and communication in SpecC

(c) Show a short code excerpt that demonstrates a non separated code and code that separates these concerns (in C/C++/SpecC/SystemC or similar).

```
Example 1 Separate Concerns

interface BasicHandshake{
      void Send(char);
      char Receive(void);
};

channel HandShake implements
BasicHandshake {
      char Data;
      event Valid, Ack;
      void Send(char _data)
      {
            Data = _data;
            notify Valid;
            wait Ack;
      }
      unsigned char Receive(void){
            char _data;
            wait Valid;
            _data = Data;
            notify Ack;
            return _data;
      }
};

behavior Sender(BasicHandshake port)
{
      char compute();
      void main(void)
      {
            char data;
            data = compute(data);
            port.Send(data);
      }
};
```

```
Example 2 Non-Separate Concerns

/* no equivalent to BasicHandshake
Interface or channel HandShake
because Computation and
Communication are concerns that are
not separated in this example. If
the communication specification were
to change, this change has to be
reflected in all behaviors adopting
this specification. This is not true
for example 1 where the change will
only be reflected in the channel
implementation with no changes to
behavior using the channel */

behavior Sender(out event req, in
event ack, out char data){
      char compute(char _data);
      void main(void)
      {
            data = compute(data);
            notify req;
            wait ack;
      }
};
```

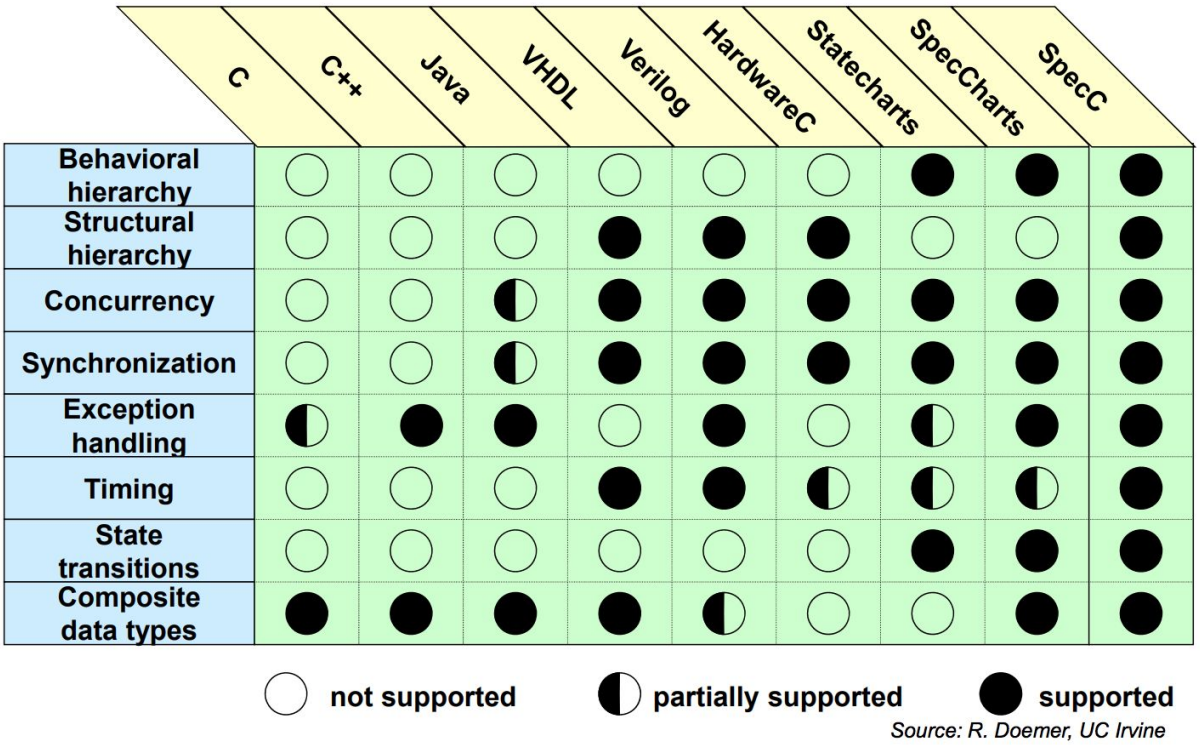
```
behavior Receiver(BasicHandshake
port)
{
      char compute(char _data)
      void main(void)
      {
            char _data;
            port.Recieve(_data);
            _data = compute(_data);
      }
};

behavior Main(void)
{
      Channel     C;
      Sender      S(C);
      Receiver    R(C);
      int main(void)
      {
            par
            {
                  S.main();
                  R.main();
            }
            return 0;
      }
};
```

```
behavior Receiver(in event req, out
event ack, in char data){
      char _data;
      char compute(char _data);
      void main(void){
            wait req;
            _data = data;
            notify ack;
            _data = compute(_data);
      }
};

behavior Main(void){
      event ack, req;
      char data;
      Receiver R(req,ack,data);
      Sender S(req,ack,data);
      int main(void)
      {
            par
            {
                  S.main();
                  R.main();
            }
            return 0;
      }
};
```

# Problem 2.2: Languages




(a) Why are sequential programming languages (e.g. C/C++/Java) considered to be insufficient for embedded system specification and design?

C/C++/Java fail are insufficient for embedded system design and specification due to their failure at modeling the following:

- Behavioral hierarchy (No composition of varied behaviors like parallel and pipelined)
- Structural hierarchy (Behaviors of Behaviors)
- Concurrency (natively without pthreads or external libraries)
- Synchronization (again without pthread synchronization primitives or external libraries)
- Timing (Via explicit language semantics)
- State machines and state transitions (Via explicit language semantics)

(b) Why are hardware design languages (e.g. VHDL/Verilog) considered to be insufficient for embedded system specification and design?

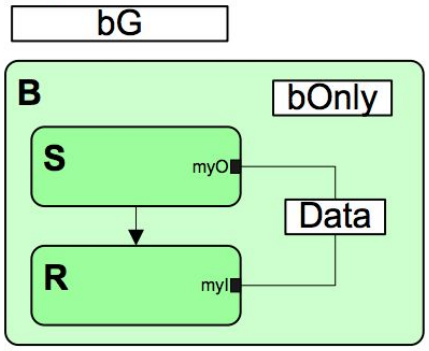
VHDL/Verilog fail are insufficient for embedded system design and specification due to their failure at modeling the following:

- Behavioral hierarchy (No composition of varied behaviors like parallel and pipelined)
- State machines and state transitions (Via explicit language semantics)

## Problem 2.3: Variables and Scope

Scope defines visibility / accessibility.
- Global Scope
- Class (Behavior / Channel) Scope
- Method local scope

- Variables exported through ports




```
1  int bG = 13;
2  behavior S(out int myO) {
3    void main(void) {
4      myO = 42;
5      //bOnly = 2; (would break)
6  }};
7  behavior R(in int myI) {
8    void main(void) {
9      int i;
10     i = myI;
11     printf("Value= %d\n", i);
12 }};
13 behavior B() {
14  int bOnly, Data;
15  S s(Data);
16  R r(Data);
17  void main(void) {
18    bOnly = 41; // not clean
19    s;
20    r;
21 }};
```

a) In the example above identify a global variable, where is it visible / accessible?

A global variable is bG, it is visible to all behaviors in the code as well as all subroutines within all behaviors.

b) Give one example of a method / function local variable and the line numbers in which this variable is visible /accessible (== scope).

Line 10 ***" int i "*** , accessed in line 12 and 13.

c) Give two examples of class (behavior / method) local variables.

***" Behavior B "*** line 16, variables ***" int bOnly "*** and ***" int Data "*** .

d) Is a class local variable visible in the children of the class (child behavior or child channel)?

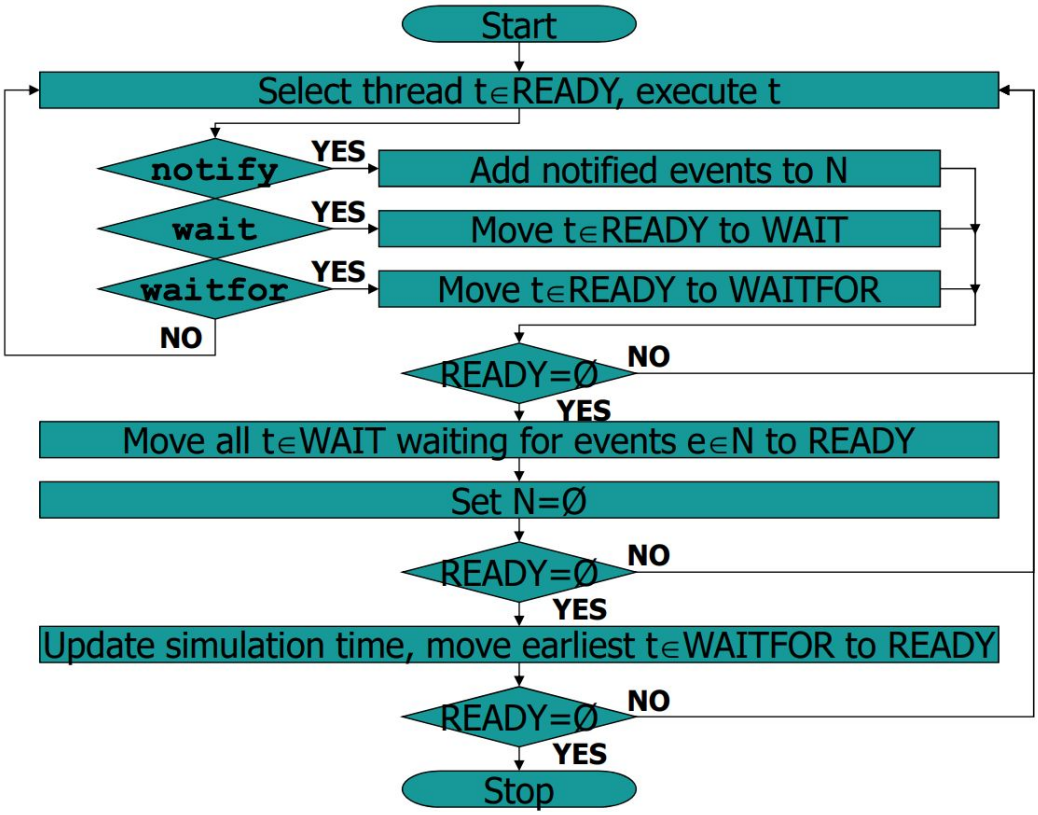
Interfaces cannot have local variables. All local variables to a behavior/channel are visible to the child methods (functions) of said class. However, local variables need to be explicitly passed to child behaviors/ channels via ports and are not visible by default to instantiated sub behaviors/ channels.
e) How is the variable Data accessed inside the behavior R? How does R get the access to this variable? List the chain that leads to the variable itself.

Variable Data is accessed in behavior R as an input ***int*** listed in it's port list, so it can only read the variable and never write to it. Behavior S accesses variable D as an output int list in it's port list and as a result it can only write to the variable and never read from it. Behavior R gets access to the variable after behavior S executes due to the sequential behavioral composition of S and R in behavior B. The chain of events is as follows:

1. Behavior B calls the main function of sub behavior S
2. Main function of behavior S writes the value 42 to the variable Data
3. Behavior S concludes with the end of its main function
4. Behavior R calls the main function of sub behavior R
5. Data is read by behavior R and is printed to console via a printf system call
6. Behavior R concludes with the end of its main function

# Problem 2.3: Discrete-Event Language Semantics

Disclaimer: Variations in what the scheduler picks to run first for the behaviors will only be explicitly mentioned if they affect the validity of the code. Only changes in output will be noted.




a)

```
behavior A(event x,
           event y)
{
  void main(void)
  {
    printf("A\n");
    notify x;
    wait y;
    printf("A end\n");
  }
};
```

```
behavior B(event x,
           event y)
{
  void main(void)
  {
    printf("B\n");
    notify y;
    wait x;
    printf("B end\n");
  }
};
```

```
behavior Main(void) {
  event x, y;

  A a(x,y);
  B b(x,y);

  int main(void) {
    par { a; b; }
    printf("Done\n");
    return 0;
  }
};
```

**Outputs in order of printfs executed**

Assuming A is picked from the ready queue first, reverse if B is picked

1) A

2) B

Assuming A is picked from the ready queue again, reverse if B is picked

3) A
4) B
5) Done

**Sequences of behavior executions**

Assume A is always picked to run first

1) A and B are both in the ready queue
2) A executes prints A
3) A issues a notify for X, X moved to N
4) A encounters a wait statement and is moved to the WAIT queue
5) B only behavior in READY queue, so runs and issues a notify for Y
6) Event Y is collected in the event queue N
7) B encounters a wait and is moved to moved to the WAIT queue
8) READY empty, new delta cycle is created
9) All behaviors waiting for events in Wait queue are matched with events in N
10) A and B moved to READY queue
11) A executes first and prints A then terminates
12) B executes and prints B then terminates
13) Main prints Done then terminates

**Opportunities for Parallelism**

Steps 2,3,4 in parallel with 5, 6, 7 as well as, 11 in parallel with 12

**Termination**

The program terminates normally with no deadlocks

(b)

```
behavior A(event x,
           event y)
{
  void main(void)
  {
    printf("A\n");
    wait x;
    wait y;
    printf("A end\n");
  }
};
```

```
behavior B(event x,
           event y)
{
  void main(void)
  {
    printf("B\n");
    notify x;
    notify y;
    printf("B end\n");
  }
};
```

```
behavior Main(void) {
  event x, y;

  A a(x,y);
  B b(x,y);

  int main(void) {
    par { a; b; }
    printf("Done\n");
    return 0;
  }
};
```

**Outputs in order of printfs executed**

Assume A is always picked to run first when possible

1) A
2) B
3) B end

**Sequences of behavior executions**

Assume A is always picked to run first when possible, deadlock occurs regardless if B is picked

1) A and B are in READY, A picked to run
2) A prints A
3) A waits on event X, so moved to WAIT queue
4) B on READY queue, so runs
5) B prints B
6) B adds event X to N queue
7) B adds event Y to N queue
8) B prints B end
9) B terminates
10) READY empty, nothing to run
11) N queue matched with WAIT and A is moved to READY
12) N cleared, event Y lost
13) A picked from READY to run
14) A waits for y, so moved to WAIT queue
15) READY empty, nothing to run
16) Deadlock, A waiting on EVENT not in N

**Opportunities for Parallelism**

No opportunities for parallelism

**Termination**

The program terminates does not terminate due to a deadlock, possible fix:

```
behavior A(event x, event y)
{
	void main(void)
	{
		printf("A\n");
		wait x;
		notify y;
		wait y;
		printf("A end\n");
	}
};

behavior Main(void)
{
	event x,y;

	A a(x,y);
	B b(x,y);
	int main(void)
	{
		par
		{
			a;
			b;
		}
		printf("Done\n");
		return 0;
	}
};
```

```
behavior B(event x, event y)
{
	void main(void)
	{
		printf("B\n");
		notify x;
		notify y;
		printf("B end\n");
	}
};


/*
A can self notify to release itself
from the deadlock and since it does
that right before it waits, the
event is never lost. The first
notify y from B is lost no matter
what though.
*/
```

(c)

```
behavior A(event x,
           event y)
{
  void main(void)
  {
    printf("A\n");
    notify x;
    wait x;
    printf("A end\n");
  }
};
```

```
behavior B(event x,
           event y)
{
  void main(void)
  {
    printf("B\n");
    notify x;
    wait x;
    printf("B end\n");
  }
};
```

```
behavior Main(void) {
  event x, y;

  A a(x,y);
  B b(x,y);

  int main(void) {
    par { a; b; }
    printf("Done\n");
    return 0;
  }
};
```

Gunar Schirner 3

**Outputs in order of printfs executed**

Assume A is always picked to run first when possible

1) A
2) B
3) A end
4) B end
5) Done

**Sequences of behavior executions**

Assume A is always picked to run first when possible

1) A and B are in READY, A picked to run
2) A runs and prints A
3) A notifies event X and X is moved to N queue
4) A waits on X and is moved to WAIT queue
5) B on READY queue so it runs
6) B prints B
7) B notifies event X, X is moved to N queue but since it's already there nothing happens
8) B waits on X
9) READY queue empty
10) WAIT queue matched with N queue
11) A and B moved to READY because of event X
12) N queue cleared
13) A is picked to run
14) A prints A end
15) A terminates
16) B on READY, so runs
17) B prints B end
18) B terminates
19) Main prints Done

**Opportunities for Parallelism**

Steps 1, 2, 3, 4, 12, 13, 14 and 5, 6, 7, 8, 15, 16, 17 can run in parallel due to each behavior self notifying

**Termination**

The program terminates with no deadlock

(d)

```
behavior A(event x,
           int f)
{
  void main(void)
  {
    if(f) wait x;
    f += 1;
    // critical sec.
    if(f>1) exit(1);
    notify x;
    f -= 1;
  }
};
```

```
behavior B(event x,
           int f)
{
  void main(void)
  {
    if(f) wait x;
    f += 1;
    // critical sec.
    if(f>1) exit(1);
    notify x;
    f -= 1;
  }
};
```

```
behavior Main(void) {
  event x;
  int f = 0;

  A a(x,f);
  B b(x,f);

  int main(void) {
    par { a; b; }
    printf("Done\n");
    return 0;
  }
};
```

**Outputs in order of printfs executed**

Assume A is always picked to run first when possible

1) Done

**Sequences of behavior executions**

Assume A is always picked to run first when possible

1) A and B are in READY, A picked to run
2) F = 0, so condition fails no wait
3) F incremented by 1
4) F = 1 so condition fails no exit
5) A notifies X and X is moved to N
6) F = 0
7) A terminates
8) B on READY queue so runs
9) F = 0 so condition fails no wait
10) F = 1
11) F = 1 so condition fails no exit
12) B notifies X and X is moved to N, but already there so no change
13) F = 0
14) B terminates
15) Main prints Done

**Opportunities for Parallelism**

No opportunities for parallelism because if A and B execute truly concurrently, Only one behavior will be able to get to the critical section of the code and the other (assuming the increment is atomic) will exit. This will break the code and prevent on behavior from entering the critical section.

**Termination**

The program terminates with no deadlock

(e)

```
behavior A(int myA, event e) {
  void main(void) {
    myA = 10;
    notify e;
    myA = 11;
  }
};
behavior B(int myA, event e) {
  void main(void) {
    wait e;
    myA = 42;
  }
};
behavior Main(void){
  int myA; event e;
  A a(myA, e);
  B b(myA, e);

  int main(void){
    par { a; b; }
    printf("myA: %d\n", myA);
    return 0;
  }
};
```

**Outputs in order of printfs executed**

Assume A is always picked to run first when possible

1) myA: 42

**Sequences of behavior executions**

Assume A is always picked to run first when possible

1) A and B are in READY, A picked to run
2) A sets myA to 10
3) A notifies e so e moved to N
4) A sets myA to 11
5) A terminates
6) B on READY so runs
7) B waits on e
8) READY empty
9) WAIT matched with N, B moved to ready
10) N cleared
11) B sets myA to 42
12) B terminates
13) Main prints myA: 42

**Opportunities for Parallelism**

Steps 2,3,4,5 and 7 can happen concurrently

**Termination**

The program terminates with no deadlock

(f)

```
behavior A(int myA, event e) {
  void main(void) {
    waitfor 10;
    myA = 11;
    waitfor 10;
  }
};
behavior B(int myA, event e) {
  void main(void) {
    waitfor 11;
    myA = 10;
  }
};
behavior Main(void){
  int myA; event e;
  A a(myA, e);
  B b(myA, e);

  int main(void){
    par { a; b; }
    printf("myA: %d\n", myA);
    // now() returns current
    //       simulated time
    printf("Time: %llu\n", now());
    return 0;
  }
};
```

**Outputs in order of printfs executed**

Assume A is always picked to run first when possible

1) myA: 42

**Sequences of behavior executions**

Assume A is always picked to run first when possible

1) A and B are in READY, A picked to run
2) A encounters wait for, so moved to WAITFOR queue with absolute time 10
3) B on ready queue so runs
4) B encounters wait for, so moved to WAITFOR with absolute time 11
5) READY queue empty
6) No behaviors waiting for events, and N queue is empty
7) Simulation updated to lowest in WAITFOR, so set to 10
8) A moved to READY queue so runs
9) A sets myA to 11

10) A encounters waitfor so moved to WAITFOR with absolute time 20
11) No behaviors waiting for events, and N queue is empty
12) Simulation updated to lowest in WAITFOR, so set to 11
13) B moved to READY, so runs
14) B sets myA to 10
15) B terminates
16) No behaviors waiting for events, and N queue is empty
17) Simulation updated to lowest in WAITFOR, so set to 20
18) A moved to READY, so runs
19) A terminates
20) Main prints myA: 10
21) Main prints Time: 20

**Opportunities for Parallelism**

Behavior A and B can occur concurrently provided simulation time semantics are preserved

**Termination**

The program terminates with no deadlock

## Problem 2.3 Part A

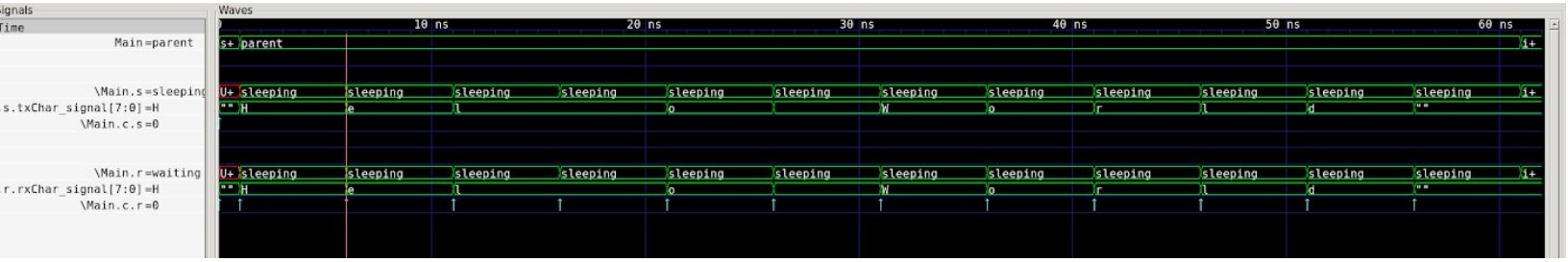
```
QUEUE TEST STARTING!
Receiver starting...
Sender starting...
Sending 'H'
Sending 'e'
Sending 'l'
Sending 'l'
Sending 'o'
Sending ' '
Received 'H'
Received 'e'
Received 'l'
Received 'l'
Received 'o'
Sending 'W'
Sending 'o'
Sending 'r'
Sending 'l'
Sending 'd'
Received ' '
Received 'W'
Received 'o'
Received 'r'
Received 'l'
Received 'd'
QUEUE TEST DONE!

/*
According to specc execution
semantics, the sender keeps sending
until blocked because it hits the
queue size. At that point receiver
runs and begins receiving data, it
is only blocked when the queue is
empty. When that occurs, the sender
runs again and so … the cycle
repeats itself.
*/
```

```
DOUBLE HANDSHAKE STARTING!
Receiver starting...
Sender starting...
Sending 'H'
Sending 'e'
Received 'H'
Sending 'l'
Received 'e'
Sending 'l'
Received 'l'
Sending 'o'
Received 'l'
Sending ' '
Received 'o'
Sending 'W'
Received ' '
Sending 'o'
Received 'W'
Sending 'r'
Received 'o'
Sending 'l'
Received 'r'
Sending 'd'
Received 'l'
Received 'd'
DOUBLE HANDSHAKE TEST DONE!
MAIN DONE!
/*
Unlike the queue, when sender runs,
it is blocked immediately after the
first character and yields control
to the receiver. The receiver then
runs and receives the character then
is blocked again because the shared
variable between sender and receiver
is empty. The cycle also repeats
itself
*/
```

# Problem 2.3 Part B

1) **Configuration:**
   Sender waitfor 5
   Reciever waitfor 5
   Queue size 5

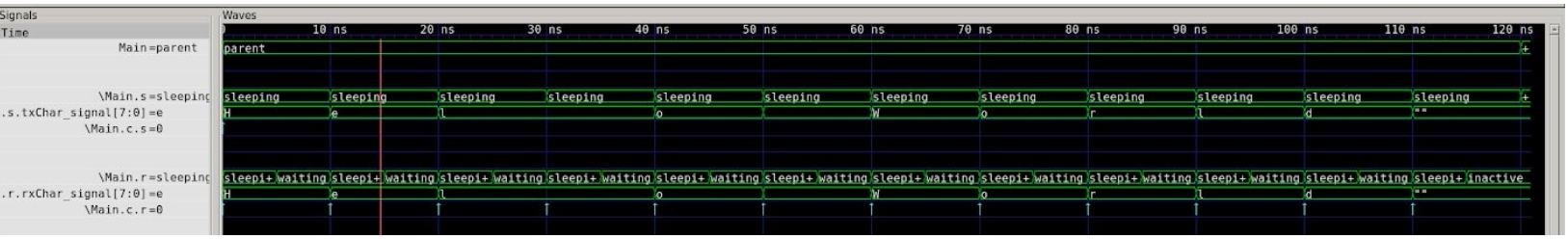




Due to the equal rate of consumption and production, the queue can never be exhausted. As a result, the behavior is similar to the double handshake untimed model in terms of sequence of consumption and production. However, the sequence in the untimed model depends on the logical ordering of waits and notifies of events. In this timed model sequence depends also on waits and notifies but waitfors space out the sequence. In the lower trace it can be shown that the logical ordering expected for consumption and production is followed. In the first delta cycle at time 1ns, the sender sends the first character to the queue and hits a waitfor. This happens in atomic time at 1ns. Receiver encounters the read from queue and at the first delta cycle the queue is still empty. READY queue becomes empty and a new delta cycle is created. Receiver is released by the queue (as shown with the rising edge on the Main.c.r event). It then acquires the character and encounters a waitfor and thus it is parked. The behaviors wake up at 2ns and continued this pattern until all characters are transmitted. It must be noted that the 1ps time shifts denote delta cycles and they have no effect on the waitfors. Absolute time is calculated without taking into account delta cycles. It must also be noted that the assignment to the signals seems to occur at the end of a delta cycle, which may introduce some confusion.
The above traces begin at 1ns because a waitfor(1) was added to the main behavior. This has been removed because the other figures don't go down to the ps scale.

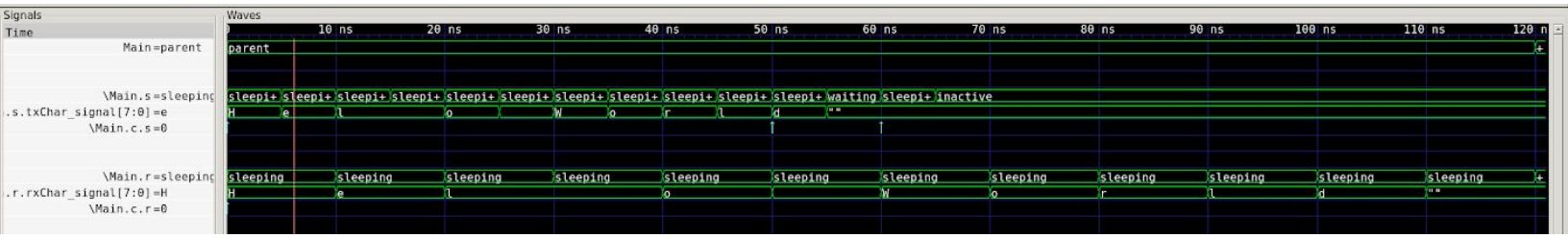
**2) Configuration**

Sender 10
Receiver 5
Queue 5




In this configuration, the receiver is starved and thus enters long periods of waits on the channel and is only woken up when sender wakes up from the waitfor statement and pushes data into the queue. The queue cannot be exhausted here due to the rate of consumption being higher than production. The queue acts as a modulator for consumption rate due to blocking recieve.

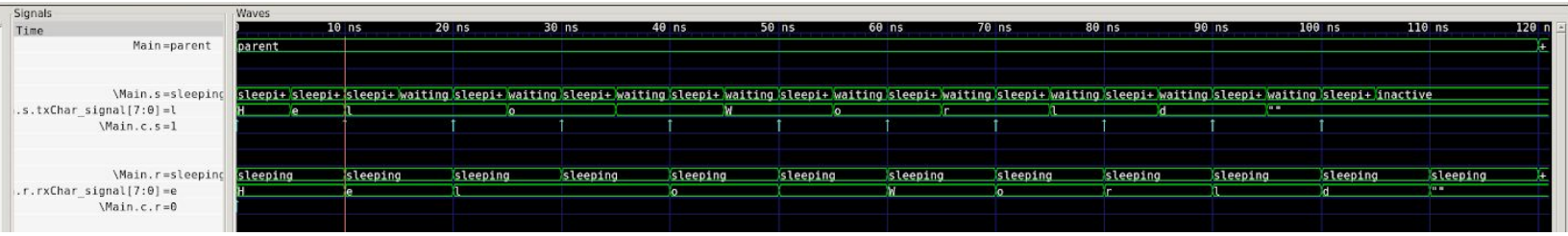
3) **Configuration**
   Sender 5
   Receiver 10
   Queue 5




This configuration is the closest to exhibiting behavior similar to the untimed queue model. The sender begins to block at the last null character to be sent. If the sending string was longer, the stalling of the sender would have been more pronounced. This situation has the capacity to exhaust the queue. It must be noted that the first element H is immediately consumed at t 0 because both threads are active at that moment in time, and the queue is empty. At that point, the sender did not have a chance to produce at twice the rate of the receiver yet. Since the standard library has queues that use blocking send and not a generic overwrite policy, it acts as a modulator for production rate and it forces production to eventually match consumption after a few cycles. It is not apparent how that happens in this configuration, but it is in the next one.

4) **Configuration**

Sender 5
Receiver 10
Queue 1




This configuration is similar to the previous one if the sent string had been longer. The sender quickly exhausts the queue of size 1 and blocks and it's rate of production is quickly reduced by blocking. After steady state production begins matching consumption. This is similar to the untimed double handshake model more so than the untimed queue model, as the double handshake is the ultimate modulator for production and consumption.