Homework 2
Design, MoC, Languages, SpecC

Assigned:	**Fri**	**10/11/19**	
Due:	**Fri**	**10/18/19,**	**12pm**

Instructions:

- Please submit your solutions via Canvas. Submissions should include a single PDF with the write-up and an archive for any supplementary files.
- You may discuss the problems and the concepts with your classmates. This fosters group learning and improves the class' progress as a whole. However, make sure to submit your own independent and individual solutions.
- Some questions might not have a clearly correct or wrong answer. In such cases, grading is based on your arguments and reasoning for arriving at a solution.

Problem 2.1: Language Concepts (15 points)

We have discussed the importance of separation of concerns.

(a) What is separation of concerns?

(b) Name two concerns that should be separated in a language as they cover orthogonal aspects?

(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).

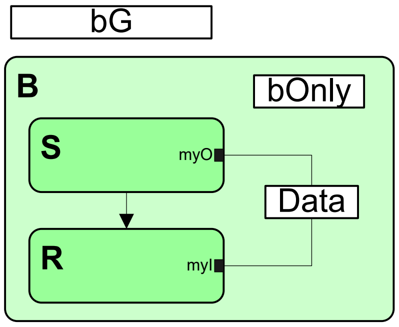
Problem 2.2: Languages (10 points)

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

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

Problem 2.3: Variables and Scope (20 points)

- 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 }};
 8 behavior R(in int myI) {
 9   void main(void) {
10     int i;
11     i = myI;
12     printf("Value= %d\n", i);
13
14 }};
15 behavior B() {
16  int bOnly, Data;
17  S s(Data);
18  R r(Data);
19  void main(void) {
20
21    bOnly = 41; // not clean
22    s;
23    r;
24 }};
```

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

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

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

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

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.

Problem 2.3: Discrete-Event Language Semantics (25)

For each of the following SpecC code examples, what are the outputs and sequences of behavior executions according to SpecC's discrete-event semantics? Show possible parallel simulation, if any. Does each program terminate normally? If not, why not, and how could the code be fixed? Note that you are free to run the examples in the SpecC simulator, but you need to provide general explanations of all possible behaviors.

(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;
  }
};
```

(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;
  }
};
```

(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;
  }
};
```

(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;
  }
};
```

(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;
  }
};
```

(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;
  }
};
```

Problem 2.3: SpecC: Standard Channel, Timing (30 points)

This assignment continues with the simple Producer-Consumer example in SpecC from previous homework. Recollect that the program should contain two parallel behaviors `S` and `R` that communicate the string "Hello world" from sender to receiver. The communication should be character by character (one byte at a time), and both behaviors should print the characters as they are sent and received to the screen. After the entire message is transmitted, both behaviors should end and the simulation should cleanly terminate.

(a) Create a copy of pd_custCh.sc and name it pd_stdCh.sc. Now replace your custom channel with (1) a `c_double_handshake` and (2) a `c_queue` instance out of the SpecC standard channel library. Use queue depths/sizes of 1 and 5 bytes. Again, compile and simulate the code. Does the program behave differently with your custom, a `c_queue` or a `c_double_handshake` channel? Explain any differences. You can inspect the code for standard channels in their source files (in the `$SPECC/import` directory) or in the debugger. For the latter, you need to add the `$SPECC/import` directory to `gdb`'s search path for source files, such that you can step into channel method calls:

```
(gdb) dir $SPECC/import
```

(b) Your SpecC simulation so far has been untimed. Turn your SpecC program now into a timed model. Simulate execution timing by adding `waitfor()` statements into `R` and `S` behaviors whenever a new character is sent or received. Furthermore, insert code to print the total simulated time at the end of the simulation. Use the model from (a) with a `c_queue` channel of size/depth 5. For each of the following cases, compile and simulate the code. Explain any differences, including a comparison to the untimed model:

1) Insert a `waitfor(5)` delay into both `R` and `S`.
2) Increase the delay in `S` to 10 time units.
3) Reduce the delay in `S` to 5 and increase the delay in `R` to 10 time units.
4) Change the queue size from 5 to 1.

For timed models, the SpecC simulator also includes the capability to create traces and waveforms of model behavior over time in standard value change dump (VCD) format. To enable tracing, compile your model with the `-Tvcds` command line option passed to `scc`. This will produce a `<design>.vcd` waveform file when running the simulation. Tracing options can be controlled by an associated `<design>.do` command file. See the examples in `$SPECC/examples/trace` (including the README file) for more details. Generated traces can then be opened and visualized in any VCD waveform viewer, such as `gtkwave` available on the LRC machines:

```
% gtkwave <design>.vcd
```

You can insert behavior instances (such as `Main.r` and `Main.s`) into the waveform display to observe their traced behavior over time. Submit such waveform plots as part of your explanations for behavior seen in (1)-(4).