Embedded System Design and Modeling

EE382N.23, Fall 2017

Homework #1

Design Languages

Assigned: September 6, 2017
Due: ~~September 18, 2017~~ September 20, 2017

Instructions:

- Please submit your solutions via Canvas. Submissions should include a single PDF with the writeup and a single Zip or Tar archive for any supplementary files (e.g. source files, which has to be compilable by simply running 'make' and should include a README with instructions for running each model).
- You may discuss the problems with your classmates but make sure to submit your own independent and individual solutions.
- Some questions might not have a clearly correct or wrong answer. In general, grading is based on your arguments and reasoning for arriving at a solution.

Problem 1.1: SpecC Language

The SpecC environment is installed on the ECE LRC Linux servers. Instructions for accessing and setting up the tools are posted on the class website:

http://www.ece.utexas.edu/~gerstl/ee382n_f17/docs/SpecC_setup.pdf

In short, once logged in, you need to load the corresponding module:

```
module load sce
```

The SpecC installation includes a comprehensive set of examples showing the features and use of the language. Examples are found in `$SPECC/examples/simple/`. You can copy them into a working directory:

```
% mkdir hw1.1
% cd hw1.1
% cp $SPECC/examples/simple/* .
```

And then use the provided `Makefile` to compile and simulate all examples:

```
% make all
% make test
```

It is recommended to inspect the sources of all examples and the included `Makefile` to understand the use of the SpecC compiler (`scc`) for the compilation and simulation process, and to experiment with the `scc` command-line usage and with the various `sir_`*`xxx`* tools. Information about all tools and `scc` is available via their man pages:

```
% man scc
% man sir_xxx
```

You can manually inspect, compile and execute an example on the command line as follows:

```
% less HelloWorld.sc
% scc HelloWorld -vv
% ./HelloWorld
```

Also practice working with the SpecC Internal Representation (SIR) and associated tools. You can compile an example into its (binary) SIR representation as follows:

```
% scc Adder -sc2sir -vv
```

You can then use the various sir_xxx tools to inspect and manipulate your design:

```
% man sir_list
% sir_list –t Adder.sir
% man sir_tree
% sir_tree –bt Adder.sir FA
```

Finally, a SIR file can be compiled into a simulation executable as follows:

```
% scc Adder –sir2out –vv
% ./Adder
```

For this assignment, you are asked to develop, simulate and debug a simple Producer-Consumer example in SpecC. 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. You program output should look like this:

```
Receiver starting...
Sender starting...
Sending 'H'
Received 'H'
Sending 'e'
Received 'e'
Sending 'l'
Received 'l'
Sending 'l'
Received 'l'
Sending 'o'
Received 'o'
Sending ' '
Received ' '
Sending 'w'
Received 'w'
Sending 'o'
Received 'o'
Sending 'r'
Received 'r'
Sending 'l'
Received 'l'
Sending 'd'
Received 'd'
```

(a) Write a program that realizes all communication via shared variables and events. You can follow the example on slide 33 of Lecture 2. Your code should look very similar, with only a few required modifications and additions to comply with the above specifications (e.g. to print output to the terminal). Compile and simulate the code to verify its correctness, and include a log of your program output in your report.

(b) Modify the example into a proper SpecC model that cleanly separates computation from communication. Follow the example on slide 34 of Lecture 2 to create a new channel that encapsulates basic communication primitives and replace all shared variables and events between `S` and `R` to exclusively use one or more instances of your custom channel. Compile and simulate the modified code to verify its correctness.

(c) SpecC programs can be debugged using the standard GNU Linux debugger (`gdb`). A nice graphical frontend for `gdb` is available as:

```
% ddd <design>
```

This will bring up the `ddd` graphical debugger, which allows you to debug you program directly at the SpecC source code level (if you prefer debugging at the level of the intermediate C++ code generated by the SpecC compiler, you can pass the `–sl` command line option to `scc` – this instructs the compiler to not create the necessary debug annotations that relate assembly and C++ to SpecC code). For example, in the `gdb` prompt of the debugger's command window, you can set breakpoints in SpecC behaviors and then start execution:

```
(gdb) b R::main
(gdb) run
```

Alternatively, you can use the debugger's graphical user interface (GUI) to open any SpecC source file (File→Open Source…), set breakpoints and then hit the Run toolbar button (Program→Run). From there, you can single step through the code, inspect variables, etc.

Modify your code from (b) to remove all 'Ack' related functionality from your custom channel, compile the code, and observe its behavior in the debugger. Explain the behavior of the modified code. Is the 'Ack' necessary? Why or why not?

(d) 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 /usr/local/packages/sce-20170901/import
```

(e) 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 (d) 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).

Problem 1.2: Discrete-Event Language Semantics

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) What is the output of the following parity generator example? For what values of *X*, *N* and *M* does the program terminate normally? What do the values of *X*, *N* and *M* need to be to maximize parallelism in a parallel simulator on a multi-core simulation platform?

```
behavior Even(
  event s, event f,
  inout bit[8] d,
  in    int    c)
{
  void main(void)
  {
    waitfor(X);
    notify(s);
    waitfor(N);
    wait(f);
    d[7] = c & 0x01;
  }
};
```

```
behavior OnesCntr(
  event s, event f,
  in    bit[8] d,
  inout int    c)
{
  void main(void)
  { int i;
    wait(s);
    waitfor(M);
    c = 0;
    for(i=0;i<7;i++)
      c += d[i];
    notify(f);
  }
};
```

```
behavior Main(void) {
  event s, f;
  bit[8] d = 42;
  int c;

  Even e(s,f,d,c);
  OnesCntr o(s,f,d,c);

  int main(void) {
    par { e; o; }
    printf("D: %d\n”,
           (int)d);
    return 0;
  }
};
```

(f)

```
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;
  }
};
```

(g)

```
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 1.3: Synchronous-Reactive Language Semantics

For the examples (a)-(c) above, what would be the outputs and sequences of behavior executions under synchronous-reactive semantics, e.g. when translated into corresponding Esterel programs as shown below? Does each program terminate normally? If not, why not, and how could the code be fixed? Note that the regular `await` statement in Esterel first `pause`s for one cycle before checking for the signal. The `await immediate` statement checks and immediately exits if the signal is already present in the initial/current instant (it is a shortcut for `present X else await X end`). Note that you are again free to run the examples in any Esterel simulator, but you need to provide general explanations of all possible behaviors.

(a)

```
module M:
signal x, y in  %// local signal
  [
    emit x; await immediate y; end
  ||
    emit y; await immediate x; end
  ];
end signal
end module
```

(b)

```
module M:
signal x, y in  %// local signal
  [
    await immediate x; await immediate y; end
  ||
    emit x; emit y; end
  ];
end signal
end module
```

(c)

```
module M:
signal x in  %// local signal
  [
    emit x; await immediate x; end
  ||
    emit x; await immediate x; end
  ];
end signal
end module
```