Embedded System Design and Modeling

EE382N.23, Fall 2019

Homework #2

Design Languages & Implementation Modeling

Assigned: October 10, 2019
Due: October 24, 2019

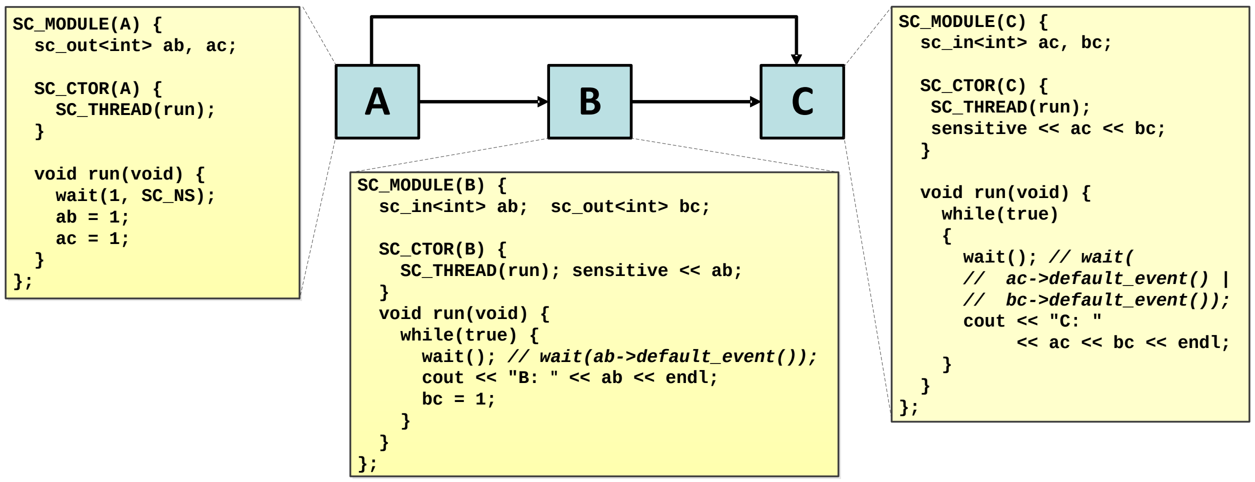
Instructions:

- 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 2.1: Discrete-Event

For each of the following SystemC examples, what is the output of the program? You are free to run the examples in the SystemC simulator, but you need to provide an explanation of all possible behaviors according to SystemC semantics.

(a) Assume that `ab`, `ac` and `bc` are `sc_signal<int>` with delta semantics that have been initialized to 0. What would be the behavior of the program if `ab`, `ac` and `bc` were instead composed out of normal variables and pure events without delta semantics (i.e. without delta variable updates and with immediate event notifications)?

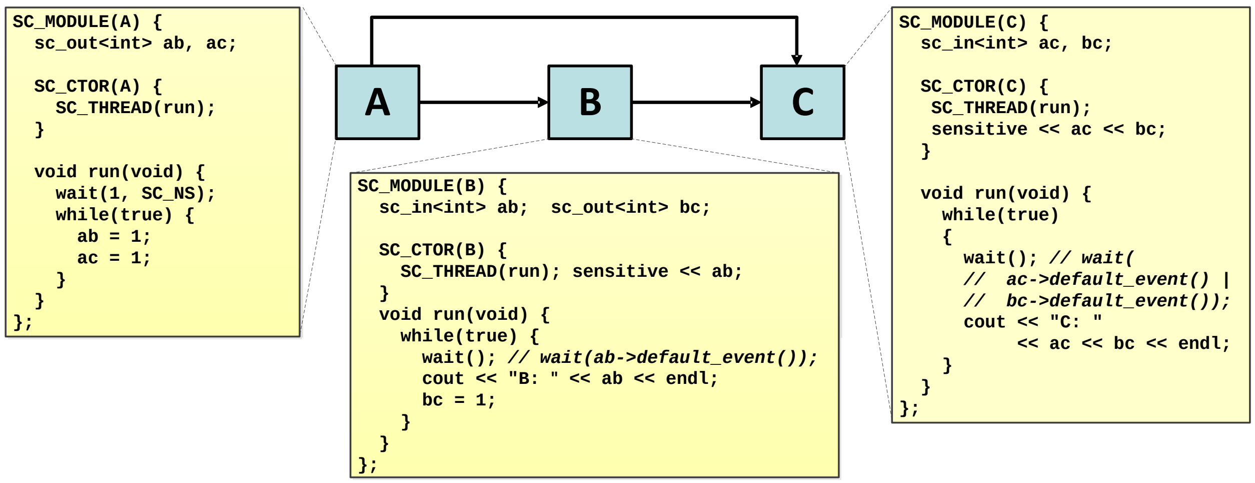


```
SC_MODULE(A) {
  sc_out<int> ab, ac;

  SC_CTOR(A) {
    SC_THREAD(run);
  }

  void run(void) {
    wait(1, SC_NS);
    ab = 1;
    ac = 1;
  }
};
```

```
SC_MODULE(B) {
  sc_in<int> ab;  sc_out<int> bc;

  SC_CTOR(B) {
    SC_THREAD(run); sensitive << ab;
  }
  void run(void) {
    while(true) {
      wait(); // wait(ab->default_event());
      cout << "B: " << ab << endl;
      bc = 1;
    }
  }
};
```

```
SC_MODULE(C) {
  sc_in<int> ac, bc;

  SC_CTOR(C) {
   SC_THREAD(run);
   sensitive << ac << bc;
  }

  void run(void) {
    while(true)
    {
      wait(); // wait(
      //  ac->default_event() |
      //  bc->default_event());
      cout << "C: "
           << ac << bc << endl;
    }
  }
};
```

(b) Assume that `ab`, `ac` and `bc` are `sc_signal<int>` that have been initialized to ‘0’. What would be the behavior of the program if `ab`, `ac` and `bc` were instead be composed out of normal variables and pure events without delta semantics/updates/notifications?

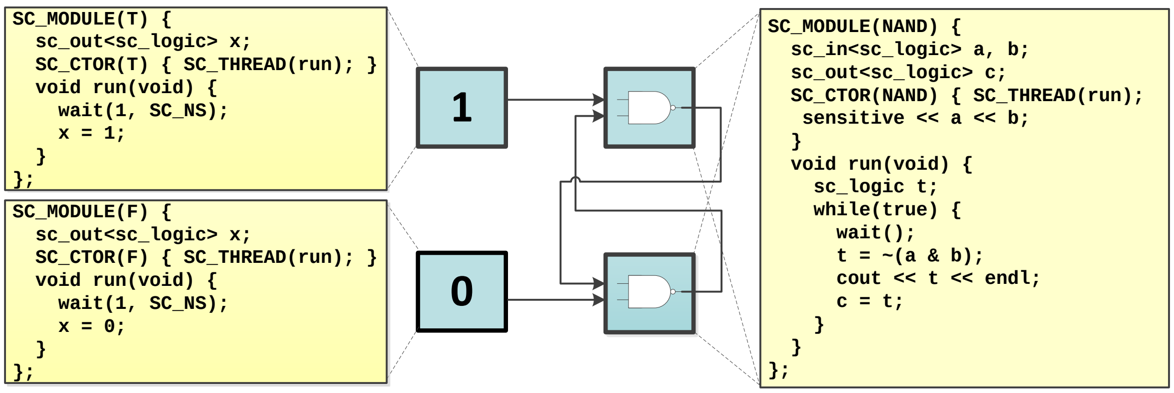


```
SC_MODULE(A) {
  sc_out<int> ab, ac;

  SC_CTOR(A) {
    SC_THREAD(run);
  }

  void run(void) {
    wait(1, SC_NS);
    while(true) {
      ab = 1;
      ac = 1;
    }
  }
};
```

```
SC_MODULE(B) {
  sc_in<int> ab;  sc_out<int> bc;

  SC_CTOR(B) {
    SC_THREAD(run); sensitive << ab;
  }
  void run(void) {
    while(true) {
      wait(); // wait(ab->default_event());
      cout << "B: " << ab << endl;
      bc = 1;
    }
  }
};
```

```
SC_MODULE(C) {
  sc_in<int> ac, bc;

  SC_CTOR(C) {
   SC_THREAD(run);
   sensitive << ac << bc;
  }

  void run(void) {
    while(true)
    {
      wait(); // wait(
      //  ac->default_event() |
      //  bc->default_event());
      cout << "C: "
           << ac << bc << endl;
    }
  }
};
```

(c) Assume that all `sc_signal<sc_logic>` have a default value of ‘X’ (unknown).

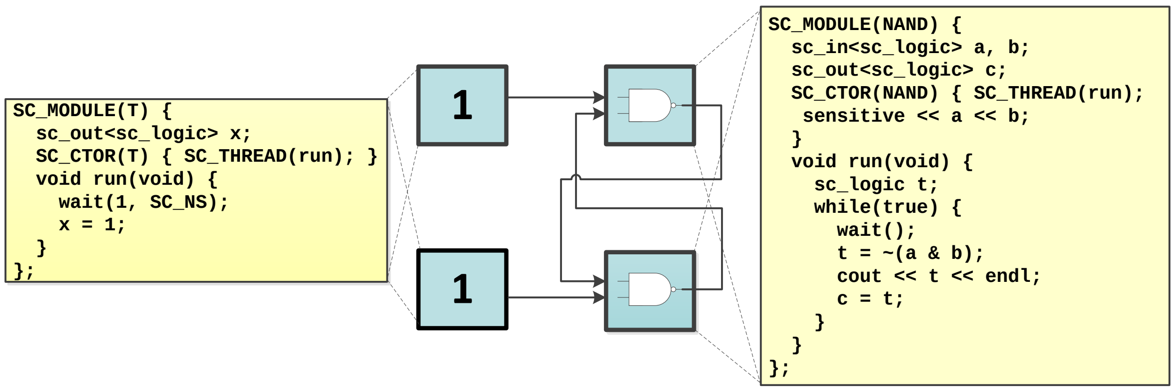


```
SC_MODULE(T) {
  sc_out<sc_logic> x;
  SC_CTOR(T) { SC_THREAD(run); }
  void run(void) {
    wait(1, SC_NS);
    x = 1;
  }
};
```

```
SC_MODULE(F) {
  sc_out<sc_logic> x;
  SC_CTOR(F) { SC_THREAD(run); }
  void run(void) {
    wait(1, SC_NS);
    x = 0;
  }
};
```

```
SC_MODULE(NAND) {
  sc_in<sc_logic> a, b;
  sc_out<sc_logic> c;
  SC_CTOR(NAND) { SC_THREAD(run);
   sensitive << a << b;
  }
  void run(void) {
    sc_logic t;
    while(true) {
      wait();
      t = ~(a & b);
      cout << t << endl;
      c = t;
    }
  }
};
```

(d) Assume that all `sc_signal<sc_logic>` have a default value of ‘X’ (unknown). What would the behavior be if signals are initialized to ‘0’ instead?



```
SC_MODULE(T) {
  sc_out<sc_logic> x;
  SC_CTOR(T) { SC_THREAD(run); }
  void run(void) {
    wait(1, SC_NS);
    x = 1;
  }
};
```

```
SC_MODULE(NAND) {
  sc_in<sc_logic> a, b;
  sc_out<sc_logic> c;
  SC_CTOR(NAND) { SC_THREAD(run);
   sensitive << a << b;
  }
  void run(void) {
    sc_logic t;
    while(true) {
      wait();
      t = ~(a & b);
      cout << t << endl;
      c = t;
    }
  }
};
```

(e) Ignoring other sources of non-determinism, e.g. coming from the C language itself, is a discrete-event model that does not allow shared variables and only supports signals for communication deterministic? If so, why? If not, what sources of non-determinism still exist?

Problem 2.2: Synchronous-Reactive

For the examples 2.1(a)-(d) above, what would be the output and behavior (sequence of executed steps) under synchronous-reactive semantics, e.g. when translated into corresponding Esterel programs as shown below? Note that Esterel supports valued signals that extend pure presence/absence into a unique value in each execution step/cycle, where `emit S(v)` emits signal `S` with value `v`, and `?S` returns the value of signal `S`. 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:
output C
signal ab, ac, bc in  %// local signal
  [
    pause; emit ab; emit ac
  ||
    loop await ab; emit bc end
  ||
    loop await [ac or bc]; emit C end
  ];
end signal
end module
```

(b)

```
module M:
output C
signal ab, ac, bc in  %// local signal
  [
    pause; loop emit ab; emit ac end
  ||
    loop await ab; emit bc end
  ||
    loop await [ac or bc]; emit C end
  ];
end signal
end module
```

(c) What sequence of values of `c0` and `c1` is emitted by this program?

```
module M:
signal i0, i1, c0, c1: boolean in  %// valued signals
  [
    pause; emit i0(true) || pause; emit i1(false)
  ||
    loop await [i0 or c1]; emit c0(not (?i0 and ?c1)) end
  ||
    loop await [i1 or c0]; emit c1(not (?i1 and ?c0)) end
  ];
end signal
end module
```

(d) What sequence of values of `c0` and `c1` is emitted by this program?

```
module M:
signal i0, i1, c0, c1: boolean in  %// valued signals
  [
    pause; emit i0(true) || pause; emit i1(true)
  ||
    loop await [i0 or c1]; emit c0(not (?i0 and ?c1)) end
  ||
    loop await [i1 or c0]; emit c1(not (?i1 and ?c0)) end
  ];
end signal
end module
```

(e) What can you say about the (non-)determinism of (valued) signals in a synchronous-reactive language?

Problem 2.3: Modeling

Given the attached SystemC code with different attempts at writing a model of a simple operating system, available at
http://www.ece.utexas.edu/~gerstl/ee382n_f19/assignments/homework2.cpp

(a) What is the behavior (output) of this program when using OS model `OS1` and `OS2`? You can run the program in the SystemC simulator, but again describe all possible behaviors according to SystemC semantics.

(b) Make minimal modifications to fix the model to represent the correct execution of tasks *A* and *B* as tasks running under control of a (most simplified/basic) custom OS with execution sequence “A1”, “B1”, “A2”, …. Make changes to task models *A* and *B*, and show any additional OS methods you need to introduce in the OSAPI as well as their implementation. What is the program output?