Rafael Fernández Ortz

Exercises for Analysis of Distributed and Concurrent Systems

Static Analysis of Distributed and Concurrent Systems

Exercise 1: Consider the following ABS codes:

```
Int main1 () {
  Fut<Int> y=a!p();
  Fut<Int> z=a!q();
  await z?;
  await y?;
  return 0;
}
```

```
Int main2 () {
  while (*) {
    Fut<Int> y = a!q();
    await y?;
    a!s();
  }
  return 0;
}
```

```
Int p () {
  a!r();
  return 0;
}
```

```
Int q () {
  y = a!r();
  awayt y?;
  return 0;
}
```

```
Int r () {
  return 9;
}
```

```
Int s () {
  return 6;
}
```

We have two initial methods `main1`, `main2`, and some internal methods (`p`, `q`, `r`) invoked from them. To clarify the code, variables `a`, `a1`, `a2` represent different ABS objects already created.

1. Write **all possible MHP pairs** (considering only method names and ignoring the *main* methods) for the three *main* methods, e.g. $p \parallel q$. Note that two concurrent executions of the same method might produce pairs of the form $p \parallel p$.

2. Compute the **local MHP** analysis information and the **MHP graph** of `main1`

1.1

main1

$p \parallel q$, $q \parallel r$, $p \parallel r$, $r \parallel r$

main2

(1S) $s \parallel q$, $s \parallel r$, $s \parallel s$

(2S) $s \parallel q$, $s \parallel q$, $s \parallel r$, $s \parallel r$, $s \parallel s$, $s \parallel s$

⋮ ...

1.2

```
1 Int main1 () {
2   Fut<Int> y=a!p();
3   Fut<Int> z=a!q();
4   await z?;
5   await y?;
6   return 0;
7 }
```

$\{\ \}$

$\{ y : \tilde{p} \}$

$\{ y : \tilde{p}, z : \tilde{q} \}$

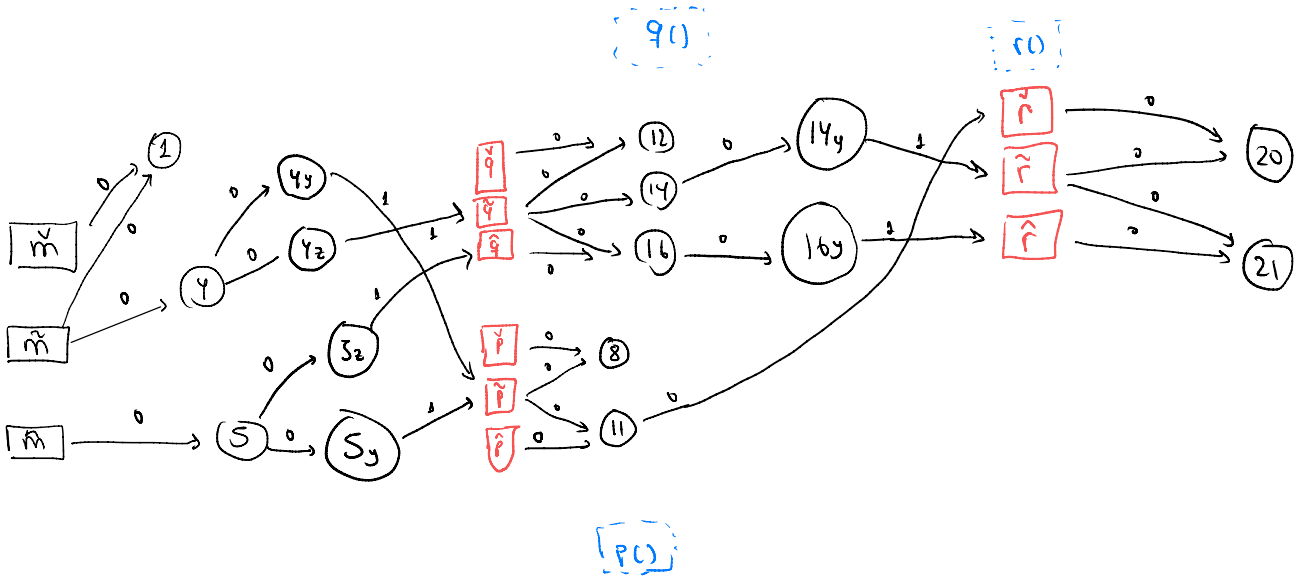
$\{ y : \tilde{p}, z : \hat{q} \}$

$\{ y : \hat{p}, z : \hat{q} \}$

```
8  Int p () {
9    a!r();
10   return 0;
11 }
```

```
12 Int q () {
13   y = a!r();
14   awayt y?;
15   return 0;
16 }
```

```
17 Int r () {
18   return 9;
19 }
```



Exercise 2: Consider the following ABS codes:

```
1  main1 () {
2    A a=new A();
3    a!blk1(a);
4  }
```

```
6
7  main2 () {
8    I a=new A();
9    I d=new D();
10   a!blk1(d);
11   d!blk1(a);
12 }
```

```
14 main3 () {
15   A a=new A();
16   B b=new B();
17   C c=new C();
18   b!blk3(c,a);
19   a!blk2(b);
20 }
```

```
22 class A implements I {
23   Unit blk1(I a) {
24     Fut<Unit> f=a!empt();
25     f.get;
26   }
27   Unit blk2(B b) {
28     Fut<Unit> f=b!empt();
29     f.get;
30   }
31   Unit empt() {}
32 }
33
34 class D implements I {
35   Unit blk1(I a) {
36     Fut<Unit> f=a!empt();
37     f.get;
38   }
39   Unit empt() {}
40 }
41
42 interface I {
43   Unit blk1(I a);
44 }
```

```
45 class B {
46   Unit blk3(C c,A a) {
47     Fut<Unit> f=c!wait(a);
48     f.get;
49   }
50   Unit empt() {}
51 }
52
53
54 class C {
55   Unit wait(A a) {
56     Fut<Unit> f=a!empt();
57     await f?;
58   }
59 }
```

As before, we have three initial methods `main1`, `main2`, `main3` and three classes `A`, `B` and `C` with their corresponding methods. Draw the dependency graphs and spot **all possible deadlocks** for the three *main* methods.

```
main1 () {
  A a=new A();
  a!blk1(a);
}
```

```
main2 () {
  I a=new A();
  I d=new D();
  a!blk1(d);
  d!blk1(a);
}
```

```
main3 () {
  A a=new A();
  B b=new B();
  C c=new C();
  b!blk3(c,a);
  a!blk2(b);
}
```

```
class A implements I {
  Unit blk1(I a) {
    Fut<Unit> f=a!empt();
    f.get;
  }
  Unit blk2(B b) {
    Fut<Unit> f=b!empt();
    f.get;
  }
  Unit empt() {}
}

class D implements I {
  Unit blk1(I a) {
    Fut<Unit> f=a!empt();
    f.get;
  }
  Unit empt() {}
}

interface I {
  Unit blk1(I a);
}
```

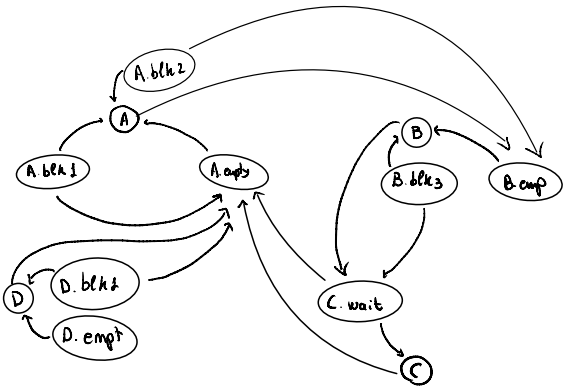
```
class B {
  Unit blk3(C c,A a) {
    Fut<Unit> f=c!wait(a);
    f.get;
  }
  Unit empt() {}
}

class C {
  Unit wait(A a) {
    Fut<Unit> f=a!empt();
    await f?;
  }
}
```

t - t

t - o

o - t

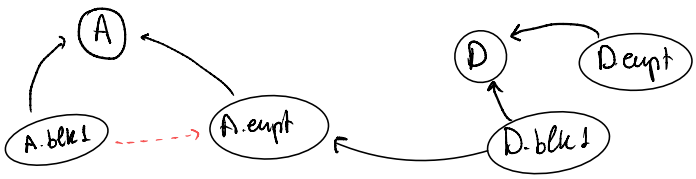


main1

A → A.blk1 → A.empty → A ; is a possible deadlock

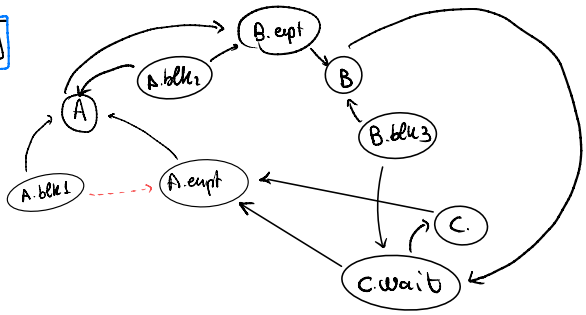
but A.blk1 ∦ A.empty ⇒ main1 is deadlock-free.

main2



⇒ main2 is deadlock-free

main3



$B.blk_3 \rightarrow B \rightarrow C.wait \rightarrow A.empty \rightarrow A \rightarrow B.empty \rightarrow B$

$A.blk_2 \rightarrow A \rightarrow B.empt \rightarrow B \rightarrow C.wait \rightarrow A.empt \rightarrow A$

There is a deadlock.

$B.blk_3.fget \parallel A.blk_2.fget$

Exercise 3: Consider the following ABS codes:

```
1  class D {
2      Int m(D b,D c){
3          Fut<Int> f;
4          f=b!n(c);
5          f.get;
6
7          Fut<Int> g;
8          g=c!n(b);
9          return g.get;
10     }
11     Int p(){
12         return 1;
13     }
14     Int n(D c){
15         Fut<Int> f;
16         f=c!p();
17         return f.get;
18     }
19     Unit exec (Di o1, Di o2, Di o3){
20       Fut<Int> f;
21       f=o1!m(o2,o3);
22     }
23 }
24 class Factory {
25     Di createD () {
26         return new D();
27     }
28 }
29
30 { // Main method
31     FactoryI f1 = new Factory();
32     FactoryI f2 = new Factory();
33     FactoryI f3 = new Factory();
34
35     Di o1 = f1.createD();
36     Di o2 = f2.createD();
37     Di o3 = f3.createD();
38
39     Di mm = new D();
40     mm.exec(o1,o2,o3);
41 }
```

Considering the analyses studied in the lecture, that is, MHP, Deadlock and Points-to analysis, please:

1. Identify a possible deadlock cycles (ignoring the MHP information) by drawing the deadlock dependency graph including the points-to information ($k = 1$) with o_l where l is the allocation site (using the number of the line of code where the object is created).

2. Now, increments the precission of the points-to analysis to $k = 2$ and include the MHP information to the deadlock analysis. Is the cycle a "feasible cycle" of it can be discarded by using the new information information?