Deadlock Analysis (SACO)

http://abs-models.org/deadlock-saco-tutorial/

Samir Genaim
Pablo Gordillo

ENVISAGE
http://www.envisage-project.eu/

Introduction

In general, deadlock situations are produced when a concurrent program reaches a state in which one or more tasks are waiting for each other termination and none of them can make any progress. In ABS, the combination of non-blocking and blocking mechanisms to access futures may give rise to complex deadlock situations and a rigorous formal analysis is required to ensure deadlock freedom.

In this tutorial we show how to use the deadlock analyzer available in EasyInterface. In Section 1 we present a small description of deadlock analysis and its parameters. In Section 2 we present how the user can execute the analyzer available in EasyInterface to check if a program is deadlock-free.

Contents

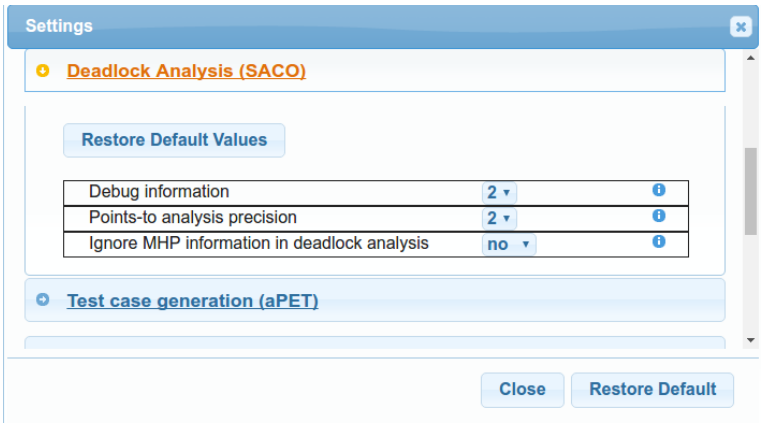
1 Deadlock Analysis **3**

2 Step by Step Example **3**

3 Further Reading **5**

1 Deadlock Analysis

In what follows we present how to use EasyInterface with the deadlock analysis. We first show how to start to use deadlock analysis within the Envisage collaboratory. For this, we must first select the analysis in the top-left pull-down menu ("Deadlock Analysis (SACO)"), and, for executing the analysis, we click on Apply. The Clear button removes all previous results. The parameters of the selected analysis are set to their default values. Nevertheless, they can be configured by clicking on the Settings button located in the top-left corner of the EasyInterface page. If you click on Settings a pop-up window appears and shows the configuration that allows us to set up the parameters of the analysis:



- *Debug Information*: Sets the verbosity of the output. The higher the number, the more verbose the output.
- *Points-to analysis precision*: The higher the number, the more detailed the analysis.
- *Ignore MHP information in deadlock analysis*: Allow to ignore the may-happen-in-parallel information. If it is ignored, the analysis will be less precise and can increment the number of false positives.

The result of the selected analysis are presented in the console. This can be done by means of text messages, markers, highlighters in the code and interactions among them. If the analysis reports that a program is deadlock-free, then there is no execution that reaches a deadlock state. When the analysis reports a *potential* deadlock, it also provides hints on program points involved in this deadlock.

In the following section, we describe the use of deadlock analysis with two examples.

2 Step by Step Example

Let us use the analysis to study the program Deadlock.abs:

```
1  module Deadlock;
2
3  interface Ai{
4      Unit m(Ci c,Bi b);
5      Unit q();
6  }
7
8  interface Bi{
9      Unit n(Ai a);
10     Unit k();
11 }
12
13 interface Ci{
14     Unit p(Bi b);
15 }
16
```

```
17 class A implements Ai{
18     Unit m(Ci c,Bi b){
19         Fut<Unit> f;
20         f=c!p(b);
21         f.get;
22     }
23
24     Unit q(){}
25 }
26
27 class C implements Ci{
28     Unit p(Bi b){
29         Fut<Unit> f;
30         f=b!k();
31         await f?;
32     }
33 }
34
35 class B implements Bi{
36     Unit n(Ai a){
37         Fut<Unit> f;
38         f=a!q();
39         f.get;
40     }
41
42     Unit k(){}
43 }
44
45 {
46 Ai a=new A();
47 Bi b=new B();
48 Ci c=new C();
49 a!m(c,b);
50 b!n(a);
51 }
```

If you click on Apply with the default parameters, you find the result of the analysis in the console. The analysis returns the cycle with the tasks involved in the deadlock. It shows also the relation between them. In this example we obtain one deadlock. If we study which tasks provoke it, we can see in the information displayed by the console that : (i) task m from object A gets blocked at program point 21 waiting for the termination of task p from object C at program point 31, which is waiting to task k from object B too, (ii) task n from object B gets blocked at program point 39 waiting for the ending of task q from object A. This two task can also run in parallel, so we have a *potential* deadlock.

Now, try to analyze the program NoDeadlock.abs:

```
1  module NoDeadlock;
2
3  interface Ci{
4      Ci m();
5  }
6
7  interface Di{
8      Ci n(Di c);
9      Ci m();
10     Unit rrun();
11 }
12
13
14 class C implements Ci{
15
```

```
16      Ci m(){
17          return new C();
18      }
19  }
20
21  class D implements Di{
22
23      Ci n(Di c){
24          Fut<Ci> f;
25          f=c!m();
26          Ci a=f.get;
27          return a;
28      }
29
30      Ci m(){
31          return new C();
32      }
33
34      Unit rrun(){
35          Di o1=new D();
36          Fut<Ci> f;
37          Di o2=new D();
38          f=o1!n(o2);
39          f.get;
40      }
41  }
42
43
44  {
45  Di mm=new local D();
46  mm.rrun();
47  }
```

In this case, the only output that the analysis returns apart from different statistics, is that the program is *deadlock-free*.

3 Further Reading

The technical details which describe the analysis presented can be found in this paper. If you want to know more about MHP analysis, you will find it in this paper.