DSA

http://abs-models.org/dsa-tutorial/

Abel García
Elena Giachino

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

Contents

1 General Overview

The Deadlock Static Analyser (DSA) performs the static and sound deadlock analysis of ABS programs described here and here. Deadlocks may be particularly hard to detect in systems with unbounded recursion and dynamic resource creation, such as server applications, which creates an unbounded number of processes. In these systems, the interaction protocols are extremely complex and state-of-the-art solutions either give imprecise answers or do not scale.

In order to augment precision and scalability we propose a modular framework combining several techniques. We meet the scalability requirement by designing a front-end inference system that automatically extracts abstract behavioral descriptions pertinent to deadlock analysis, called *behavioural types*, from code. The inference system is *modular* because it (partially) supports separate inference of modules. The inference system mostly collects method behaviors and uses constraints to enforce consistencies among behaviors. Then a standard semiunification technique is used for solving the set of generated constraints.

Our behavioural types feature recursion and resource creation; therefore their underlying models are infinite state. Our tool is particularly precise with this kind of programs, because it implements a fixpoint algorithm, which is proven to be a decision algorithm on the model.

This tutorial will show how to use the DSA features and the kind of deadlock it is able to detect.

2 Deadlock analysis

In this section we present how to check wether an ABS program is deadlock-free.

First, select "Deadlock Analysis (DSA)" from the pull-down menu at the top of the window on the center-left. The parameters of the selected analysis are automatically set, so there is nothing to be configured in the Settings section in the top-left corner.

As an example, open the program UglyChain.abs:

```
1  module C;
2
3  interface Object {
4  }
5
6  class Object implements Object {
7  }
8
9  interface C {
10     Unit m(C c);
11     Unit n(C a) ;
12     Unit q() ;
13 }
14
15 class C implements C {
16         Unit m(C c){
17                 C w = new C() ;
18                 w!m(this) ;
19                 c!n(this) ;
20         }
21         Unit n(C a){
22                 Fut<Unit> x = a!q() ;
23                 x.get ;
24         }
25         Unit q(){
26         }
27
28 }
29
30
31 {
```

```
32          C a = new C() ;
33          C b = new C() ;
34          Fut<Unit> x = a!m(b) ;
35  }
```

Let us analyze the program. Click on Apply to perform the analysis.
The output of the analysis is shown in the console:

```
Possibile Deadlock in main:  false
```

meaning that the program is deadlock-free.

Additional information are also given, such as the current release of the tool and the time for performing the analysis.

Consider, now, the example SchedulerChoice.abs:

```
1   module Test_Inference_d;
2
3   interface Object {
4   }
5
6   class Object implements Object {
7   }
8
9   interface C {
10      C m();
11      C n(C c);
12  }
13
14  class C implements C {
15      C m(){
16          C x = new C();
17          return x;
18      }
19      C n(C c){
20          Fut<C> fut = c!m();
21          return fut.get;
22      }
23  }
24
25
26  {
27  C i = new C();
28  C j = new C();
29  Fut<C> fut4 = i!n(j);
30  Fut<C> fut5 = j!n(i);
31
32  }
```

This is an example of possible run-time deadlock due to specific scheduler's choices.

Click on the Clear button for removing previous results. Then click Apply button for executing the analysis.

In this case, a possibile deadlock is detected:

```
Possibile Deadlock in main:  true
```