EasyInterface Tutorial

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

Jesús Correas Fernández
Antonio Flores Montoya
Enrique Martín Martín
Guillermo Román Díez

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

Introduction

SACO is a a static analyzer for automatically inferring *upper/lower bounds* on the worst/best-case Resource usage (a.k.a. cost) of ABS programs. The inferred upper bounds have important applications in the fields of program optimization, verification and certification. SACO is parametric on the *cost model*, i.e., the type of cost that the user wants infer (e.g., number of steps, amount of memory allocated, amount of data transmitted, etc.), and it supports different notions of cost such as sequential, parallel, peak, etc. In this tutorial, we overview the different feature of SACO by example.

Contents

1 Resource Analysis (SACO)

In what fo llows we present how to use EasyInterface with the different analyses that SACO is able to perform with some examples.

We first show how to start to use SACO within the Envisage collaboratory. For this, we must first select the analysis in the top-left pull-down menu, 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. The results of the selected analysis are presented in the console. This can be done by means of graphs, text messages, markers, highlighters in the code, and interactions among them. In the following, we describe the use of SACO by analyzing several examples.

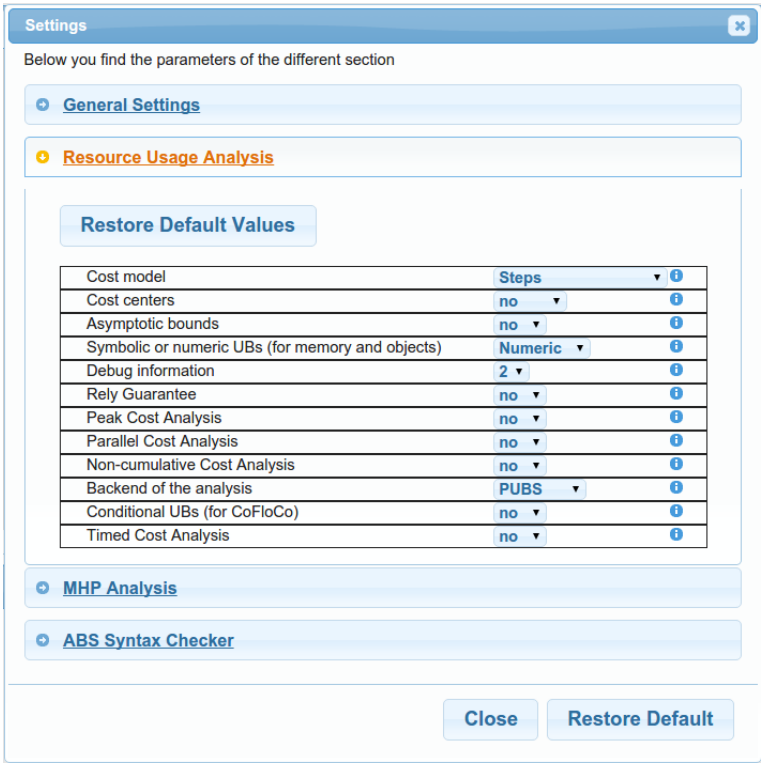
1.1 Basic Resource Analysis

Let us start by performing the basic resource analysis computed by SACO and described in this paper. To do it, open the file `VendingMachine_init.abs` , which contains the following code:

```
1  module VendingMachine_init;
2
3  interface VendingMachine {
4    Unit insertCoin();
5    Unit insertCoins( Int nCoins );
6    Int retrieveCoins();
7  }
8
9  interface PrettyPrinter {
10   Unit showIncome( Int nCoins );
11   Unit showCoin();
12 }
13
14 class IVendingMachine( Int coins, PrettyPrinter out ) implements VendingMachine{
15   Unit insertCoin(){
16     coins = coins + 1;
17   }
18
19   Unit insertCoins( Int nCoins ){
20     while( nCoins > 0 ){
21       nCoins = nCoins - 1;
22       Fut<Unit> f = this ! insertCoin();
23       await f?;
24     }
25   }
26
27   [coins < max(coins)]
28   Int retrieveCoins(){
29     Int total = 0;
30
31     while( coins > 0 ){
32       coins = coins - 1;
33       Fut<Unit> f = out ! showCoin();
34       //await f?;
35       total = total + 1;
36     }
37     return total;
38   }
39 }
40
41
```

```
42 class IPrettyPrinter implements PrettyPrinter{
43   Unit showIncome( Int nCoins ){ /*Show something*/ }
44   Unit showCoin(){ /*Show something*/ }
45 }
46
47 class IMain {
48   Unit main( Int n ){
49     PrettyPrinter o = new IPrettyPrinter();
50     VendingMachine v = new IVendingMachine( 0, o );
51
52     v ! insertCoins(n);
53     Fut<Int> f = v ! retrieveCoins();
54     await f?;
55     Int total = f.get;
56     o ! showIncome( total );
57   }
58 }
```

By selecting Resource Analysis and clicking on Settings a pop-up window appears and shows the configuration that allows us to set up the parameters for the analysis. Set the parameters as shown in this figure:



Cost model: The cost model indicates the type of resource that we are interested in measuring. The user can select among the following cost metrics: termination (only termination is proved), steps (counts the number of executed instructions), objects (counts the number of executed new instructions), tasks (counts the number of asynchronous calls to methods), memory (measures the size of the created data structures), data transmitted (measures the amount of data transmitted among the distributed objects), user-defined model (allows to write annotations in the code of the form [cost == expr] and the analysis accumulates the cost specified by the user in expr every time this program point is visited).

- *Cost centers*: This option allows us to decide whether we want to obtain the cost per cost center (i.e., for each of the abstract objects inferred by the analysis) or a monolithic expression that accumulates the whole computation in the distributed system. The value no refers to the latter case. If we want to separate the cost per cost center, we have again two possibilities. The option class shows the cost of all objects of the same class together, while object indicates the cost attributed to each abstract object.

- *Asymptotic bounds*: Upper bounds can be displayed in asymptotic or non-asymptotic form. The former one is obtained by removing all constants and subsumed expressions from the non-asymptotic cost, only showing the complexity order.
- *Symbolic or numeric*: Next, if the cost model is memory or objects, the upper bounds can be shown either *symbolically*, in terms of symbolic sizes (we use size(A) to refer to the size of an object of type A), or *numeric*, by assigning a predefined measure to them.
- *Debug*: sets the verbosity of the output (the higher the number, the more verbose the output).
- *Rely Guarantee*: performs the resource analysis taking into account the possible interleavings in the tasks execution (as described in this paper).
- *Peak Cost Analysis*: computes the peak cost analysis for all objects which are identified (see this paper).
- *Parallel Cost Analysis*: computes the parallel cost analysis of the program (see this paper).
- *Non-cumulative Cost Analysis*: computes the non-cumulative cost of the program (see this paper).
- *Backend of the Analysis*: SACO uses *PUBS* or *CoFloCo* as backend to solve the cost equations (see this technical report).
- *Conditional UBs*: computes a set of *conditional upper bounds* according to some conditions on the input parameters (see this technical report).
- *Timed Cost Analysis*: computes the cost analysis in time (see this technical report).

Let us analyze the program `VendingMachine_init.abs` with the default values, except for the asymptotic bounds parameter that must be set to yes. Click on Refresh Outline and select the entry method (method main of class IMain) in the Outline (the region on the right of the page. Then click on Apply to perform the analysis.

It can be seen in the resource analysis results yield by SACO that the upper bound is linear and it is a function on n (the input parameter of main) and on the maximum value that field coins can take, denoted max(coins). Variable n is wrapped by function nat previously defined to avoid negative costs. The upper bound is shown in the console view and also at the method's header when the mouse passes over the marker (see L48 in the program).

Now, let us analyze the main method of the file `VendingMachine.abs` , which contains the following code:

```
1  module VendingMachine;
2
3  interface VendingMachine {
4    Unit insertCoin();
5    Unit insertCoins( Int nCoins );
6    Int retrieveCoins();
7  }
8
9  interface PrettyPrinter {
10   Unit showIncome( Int nCoins );
11   Unit showCoin();
12 }
13
14 interface Main{
15   Unit main( Int n );
16 }
17
18
19 class IVendingMachine( Int coins, PrettyPrinter out ) implements VendingMachine{
20   Unit insertCoin(){
```

```
21      coins = coins + 1;
22    }
23
24    Unit insertCoins( Int nCoins ){
25      while( nCoins > 0 ){
26        nCoins = nCoins - 1;
27        Fut<Unit> f = this ! insertCoin();
28        await f?;
29      }
30    }
31
32    Int retrieveCoins(){
33      Int result = 0;
34      while( coins > 0 ){
35        coins = coins - 1;
36        Fut<Unit> f = out ! showCoin();
37        await f?;
38        result = result + 1;
39      }
40      return result;
41    }
42  }
43
44
45  class IPrettyPrinter implements PrettyPrinter{
46    Unit showIncome( Int nCoins ){ /*Show something*/ }
47    Unit showCoin(){ /*Show something*/ }
48  }
49
50  class IMain implements Main{
51    Unit main( Int n ){
52      PrettyPrinter o = new IPrettyPrinter();
53      VendingMachine v = new IVendingMachine( 0, o );
54
55      v ! insertCoins(n);
56      Fut<Int> f = v ! retrieveCoins();
57      await f?;
58      Int result = f.get;
59      o ! showIncome( result );
60    }
61  }
```

This file is just like the previous example, but including the await instruction at *L*37 that was commented out in the previous program. Analyze this program with the same configuration as before: default setting values, except for the asymptotic bounds parameter that must be set to yes. Click on Refresh Outline and select the entry method (method main of class IMain) in the Outline. Then click on Apply to perform the analysis.

The analyzer shows, by using a warning marker (see *L*51), that the resource analysis cannot infer an upper bound nor guarantee the termination of the program.

1.2 Rely-guarantee Resource Analysis

Let us now perform the rely-guarantee resource analysis, described in this paper, on the main method of the `VendingMachine.abs` file. To do so, we set the option Rely Guarantee to yes and the Cost Model to termination.

After applying the analysis, it can be seen on the default console that SACO proves that all methods of the program terminate. Let us now slightly modify the example to make method insertCoins non-terminating by removing the line *L*35 with the instruction coins = coins - 1. The analysis information is displayed as follows. For each *strongly connected component* (SCC –while loops and recursive methods are basically the

SCCs in a program), the analysis places a marker in the entry line to the SCC. If the SCC is terminating (e.g., *L*25), by clicking on the marker, the lines that compose this SCC are highlighted in yellow. On the other hand, if the SCC is non-terminating (*L*34), by clicking on the marker, SACO highlights the lines of the SCC in blue. Besides the markers, the list of all SCCs of the program and their computed termination results are printed by SACO on the console.

At this point, let us perform the rely guarantee resource analysis to infer the cost of the program. Restore the original code of *L*35, click on Settings and select the Steps cost model with the option Rely guarantee set to yes. Then click on Apply to perform the analysis.

The resulting upper bound is a function in terms of n (the input parameter of main) and in terms of the maximum value that the field coins can take, denoted by max(coins). We can observe that the cost of main is linear with respect to both. In addition, SACO shows a marker to the left of each method header to display their corresponding upper bounds.

1.3 Load Balance

At this point, let us use the resource analysis to study the load balance of the program `Performance.abs` , which contains the following code:

```
1  module Parallel;
2  import * from ABS.StdLib;
3
4  interface I {
5          Unit m (Int n);
6          Unit p (Int n, I x);
7          Unit m2 (Int n);
8          Unit q ();
9  }
10
11 class C implements I{
12         Unit m (Int n) {
13                 I a = new C();
14                 while (n > 0) {
15                         a!p(n, a);
16                         n = n - 1;
17                 }
18         }
19
20         Unit mthis (Int n) {
21                 I a = new C();
22                 while (n > 0) {
23                         a!p(n, this);
24                         n = n - 1;
25                 }
26         }
27
28         Unit p (Int n, I x) {
29                 while (n > 0) {
30                         x!q();
31                         n = n - 1;
32                 }
33         }
34
35         Unit m2 (Int n) {
36                 while (n > 0) {
37                         I a = new C ();
38                         a!p(n, a);
39                         n = n - 1;
40                 }
41         }
```

```
42
43          Unit q () {
44                  skip;
45          }
46
47  }
```

We start by applying the Resource Analysis and selecting in Settings the option Cost Centers to object. As the concurrency unit of ABS is the object, it uses the cost centers to assign the cost of each execution step to the object where the step is performed. Then click on Refresh Outline and select the method C.m on the right region of the page. Finally, click on Apply to perform the analysis. SACO returns as a result the cost centers in the program, one cost center labelled with [12] which corresponds to the object that executes C.m, and another one labelled with [13,12], which abstracts the object created at $L13$. The labels of the nodes contain the program lines where the corresponding object is created. That is, the node labeled as [13,12] corresponds to the C object, created at $L13$ while executing the main method, node identified by $L12$. In addition, SACO shows a graph with both nodes in the Console Graph view. By clicking on the node [12], SACO shows a dialog box with the upper bound on the number of steps performed by such node. Similarly, by clicking on the node [13,12], it shows the number of steps that can be executed by the object identified with [13,12]. We can observe that the node [12] performs a number of steps that is bounded by a linear function on the input parameter n, while in the node [13,12] the number of steps is bounded by a quadratic function on n. If we analyze method C.m, the cost is distributed in a different way. In this case, both nodes [20] and [21,20] have a quadratic upper bound on the number of steps performed by each node. The difference between both methods is that the call x!q() at $L30$ is performed in object [13,12] in the former case, and in object [20] in the latter.

Now we can obtain the number of instances of each object we can have in each node. Select C.m2 and unselect the previously selected methods on the Outline on the right of the page, and perform the ResourceAnalysis setting the options Cost Model to Objects and Cost Centers to Object. It can be seen in the output of SACO that the number of instances of the object identified by [37,35] is bounded by n (the input argument of method mthis). Finally, we can apply the ResourceAnalisis to C.m2 selecting Cost Model to Steps to obtain the results of the analysis for this method regarding the number of steps.

1.4 Transmission data sizes

Now, let us perform the transmission data size analysis to the following code:

```
1  module DemoTraffic;
2  import * from ABS.StdLib;
3
4  interface II {
5          Unit work (Int n, List<Int> l);
6  }
7
8  interface IS {
9          Int process (List<Int> l);
10 }
11
12 class Master (IS s) implements II {
13
14
15         Unit work (Int n, List<Int> l){
16                 while (n>0) {
17                         l = Cons(1,l);
18                         Fut<Int> q = s!process(l);
19                         q.get;
20                         n = n - 1;
21                 }
22         }
23
```

```
24 }
25
26 class Slave () implements IS{
27         Int process (List<Int> l) {return 1;}
28 }
29
30 class IMain {
31         Unit main (List<Int> l, Int n) {
32                 IS s = new Slave();
33                 II m = new Master(s);
34                 m!work(n,l);
35          }
36 }
```

Open the file `DataTransmitted.abs` . To analyze this file with the transmission data size analysis, select the analysis Resource Analysis and set the option Cost Model to Traffic. Then refresh the outline and apply the analysis to the method IMain.main. When the analysis is applied, the console will show the upper bound expressions for all possible pairs of objects identified by the analysis. For example, the last line of the console output is the upper bound of the size of the data transmitted from the node [32,31] to the node [33,31], that are the Slave and Master objects created at $L32$ and $L33$, respectively. We can observe that this upper bound linearly depends on the input parameter n, which is the number of times method process in the Slave object is invoked. On the contrary, the data transmitted from the Master object [33,31] to the Slave object [32,31] is different, as the invocation contains the list l which is passed as argument to method process. In this case, such upper bound is a quadratic function on the parameter n, as the list passed as argument grows at each iteration of the loop at $L16$, and such loop iterates n times.

In addition to the console information, a graph that shows the objects creation is displayed on the Console Graph, shown next to the default console. By clicking on a node, a message outputs the UBs for all transmissions data sizes that the selected object can perform and the objects involved in such transmissions. E.g., by clicking on the node [32,31], which corresponds to the Master object, we can see the upper bounds on the data transmitted (incoming and outgoing transmissions) from this object. As before, the labels of the nodes contain the program lines where the corresponding object is created. For instance, the node labeled as [32,31] corresponds to the Master object, created at $L32$ while executing the main method, the object identified by $L31$. In such upper bounds, the cost expression c(i) represents the cost of establishing the communication.

1.5 Non-cumulative Cost

We can illustrate the analysis for computing the non-cumulative cost with the file `Noncumulative.abs` which contains the following code:

```
1  module Noncumulative;
2  import * from ABS.StdLib;
3
4  class IMain {
5          Unit main (Int s, Int n) {
6                  [x == acquire(10)]
7                  Int i= 0;
8                  [r == acquire(100)]
9                  i = 0;
10                 [r == acquire(s)]
11                 i = 1;
12                 [r == release()]
13                 i = 2;
14                 [y == acquire(n)]
15                 i = 3;
16                 [x == release()]
17                 i = 4;
18         }
```

```
19 }
```

Select the Resource Analysis and click on Settings. Restore the default values and set the option noncumulative_cost to yes. Then refresh the outline and select the method IMain.main. The results obtained after clicking on Apply show that we have two sets of program points that can lead to the maximum on the number of resources acquired, as well as their corresponding upper bound expressions. The set [$L6, L8, L10$] corresponds to the acquire instructions at lines $L6$, $L8$ and $L10$ of the program. With this set of acquire instructions, we obtain an upper bound of the number of resources that linearly depends on the input parameter s because of the acquire at $L10$. The set [$L6, L8, L14$] can also lead to the maximum number of resources acquired, if the actual value of the input parameter n is larger than s.

1.6 Peak cost analysis

Let us continue by performing the peak cost analysis to the program `VendingMachine_init.abs` . Similarly to other analyses, we first select the entry method (method main in class IMain) in the Outline View and apply the Resource Analysis with the default options with the exception of the option Peak Cost Analysis, which must be set to yes. After applying the analysis, the peak cost analysis outputs in the console. For each identified object, all possible queue configurations are shown. A queue configuration is the set of tasks that can be in the queue simultaneously. For each queue configuration, the tasks involved in the configuration are shown. In addition, the total cost associated to the configuration is displayed as well.

The analysis of the program `VendingMachine_init.abs` shows that there are two possible queue configurations for each object identified in the program. For example, for the object [49,48] one of the configurations contains tasks for methods showIncome and showCoin, and the number of steps executed by those tasks linearly depends on the value of the field coins.

As before, SACO also shows a graph where the labels of the nodes contain the program lines where the corresponding object is created. For instance, the node labeled as [49,48] corresponds to the PrettyPrinter object, created at $L49$ while executing the main method ($L48$). By clicking on a node, the queue configurations that have been identified and their costs are shown in a message.

1.7 Parallel Cost

Let us perform the parallel cost analysis described in this paper. To do so, we open the file `Parallel.abs` which contains the following code:

```
1  module Parallel;
2  import * from ABS.StdLib;
3
4  interface IX {
5          Unit p (IY y);
6  }
7
8  interface IY {
9          Unit q ();
10         Unit s ();
11 }
12
13 class X implements IX {
14         Unit p (IY y) {
15                 skip;
16                 y!s();
17                 Int method_end = 0;
18         }
19 }
20
21 class Y implements IY {
22         Unit q () {
23                 Int method_end = 0;
24         }
```

```
25          Unit s () {
26                  Int method_end = 0;
27          }
28
29  }
30
31  class IMain {
32          Unit main () {
33
34                  IX x = new X ();
35                  IY y = new Y ();
36
37                  x!p(y);
38                  skip;
39                  y!q();
40                  Int method_end = 0;
41          }
42  }
```

Now, we select the entry method IMain.main in the Outline and apply the Resource Analysis by restoring the default values and setting the option Parallel Cost Analysis to yes. The analysis results show the computed upper bound expressions obtained for all paths identified in the DFG of the program. In addition, SACO shows the number of nodes and edges of the computed DFG.

1.8 Cost Analysis in Time

Let us continue by performing the cost analysis in time to the program in `Timed.abs` , with the following code:

```
1   module Timed;
2
3   interface Job{
4           Unit start(Int dur);
5   }
6
7   class IMain{
8           Unit main(Int n){
9                   while(n>0){
10                          Job job=new Job();
11                          job!start(10);
12                          await duration(1,1);
13                          n= n−1;
14                  }
15          }
16  }
17
18
19  class Job implements Job{
20          Unit start(Int dur){
21                  while(dur>0){
22                  [Cost: 1] dur=dur−1;
23                  await duration(1,1);
24                  }
25          }
26
27  }
```

Select the entry method IMain.main in the Outline of the program. Then, select the Resource Analysis and set the Settings option Timed to yes. After applying the analysis, the output of SACO shows

1.9 CoFloCo Backend

Finally, let us analyze the program `CoFloCoExample.abs` by using CoFloCo as backend. In this case, the file contains the following code:

```
1  module Parallel;
2  import * from ABS.StdLib;
3
4
5  class C {
6          Bool nondet=False;
7
8          Unit m1 (Int i, Int dir, Int n) {
9                  while (0<i && i < n) {
10                         if (dir == 1) {
11                                 i = i + 1;
12                         }
13                         else {
14                                 i = i - 1;
15                         }
16                 }
17         }
18
19         Unit m2 (Int x, Int y, Int a, Int r) {
20                 while (x > 0 && y > 0) {
21                         if (nondet) {
22                                 x = x - 1;
23                                 y = r;
24                         }
25                         else {
26                                 y = y - 1;
27                         }
28                         suspend;
29                 }
30         }
31
32         Unit m3 (Int x, Int y, Int a) {
33                 while (x > 0) {
34                         while (y > 0 && nondet) {
35                                 y = y - 1;
36                                 suspend;
37                         }
38                         x = x - 1;
39                 }
40         }
41
42 }
```

We can select the method of interest, that is, C.m, C.m2 or C.m3 and then perform the Resource Analysis with the default options except the option Backend, which must be set to CoFloCo. After applying the analysis, SACO shows the results of the analysis of C.m with CoFloCo. Additionally, by setting the option Conditional UBs to yes, we can obtain conditional upper bounds.

2 May-Happen-in-Parallel Analysis (MHP)

ABS is a distributed language where different methods can be invoked asynchronously. Therefore, there can be several tasks executing their code at the same time. This simultaneity complicates the analysis and understanding of ABS programs, since (in general) there is a high level of parallelism. The MHP analysis alleviates this situation by computing, for each instruction, which other instructions could execute

in parallel. This information can be very interesting for developers and testers, but it is crucial for the precision of many of the SACO analysis presented in the previous section.

The result of the MHP analysis, described in this paper, is a set of pairs of instructions that could execute in parallel in any execution of the program. It is important to stress that this analysis computes an *overapproximation* of the parallelism. In other words, if a pair of instruction can execute in parallel, then they will be detected by the analysis; however, the analysis can incorrectly detect pairs of instructions that never execute in parallel when executing the program.

2.1 MHP Analysis in the Collaboratory

In order to apply the MHP Analysis in the Collaboratory, just:

- Open the ABS program to analyze.
- Select MHP Analysis in the list of analyses.
- Refresh the list of methods in the right side (button Refresh Outline) and select the entry point of the analysis. Only one method or function can be selected as entry point, and if none is selected then the main block will be considered the entry point.
- Press the Apply button.

Let us see the results of the MHP analysis in the `VendingMachine.abs` program selecting the main method from class IMain as the entry point. The analysis reports its results in two places. First, all the pairs of instructions that can be executed in parallel are printed in the console. This information is very detailed and verbose, but difficult to understand:

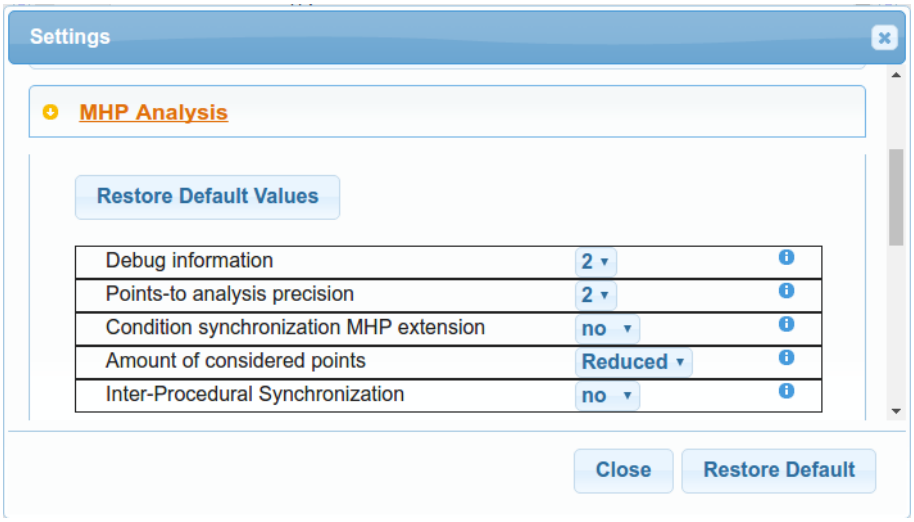
```
(...)
L28 (Await)[IVendingMachine.insertCoins(−[2,3]−)] ||
L37 (Await)[IVendingMachine.retrieveCoins(−[2,3]−)]

L20 (Entry)[IVendingMachine.insertCoin(−[2,3]−)] ||
L60 (Exit)[IMain.main(−[3,main]−)]
(...)
```

The previous fragment shows that the instruction at *L*28 (the await instruction in method insertCoins) can happen in parallel with the instruction at *L*37 (the await instruction in method retrieveCoins), and the same for instructions at *L*20 and *L*60. However, the most comfortable way of inspecting the results of the MHP analysis is by navigating the code. Every considered instruction will have a blue arrow in its left margin. When clicking the arrow of an instruction, all the other considered instructions that can happen in parallel with it will be highlighted in yellow. For example, clicking on the arrow in *L*37 will show the instructions that can happen in parallel with it, which as expected include *L*28. We have mentioned "considered instruction" because, by default, the MHP analysis only takes into account those instructions that are entry/exit points of methods and await or release instructions. In order to obtain the full list of instructions that may happen in parallel, open the settings window (button Settings, option MHP analysis) and change the option Amount of considered points from Reduced to Full. With these new settings the analysis obtains the MHP information for all the instructions in the program.

Since the MHP analysis is a sound overapproximation, instructions that cannot execute in parallel are never detected. For example, due to the await instruction at *L*57, the task retriveCoins must be finished before continuing. Therefore, the await instruction at *L*37, which is inside retriveCoins, cannot happen in parallel with the instructions after the await (*L*58 and *L*59) or the method showIncome that is invoked at *L*59. If we execute a full MHP analysis and select the arrow at *L*37, those fragments will not be highlighted.

2.2 MHP Settings

In addition to the Amount of considered points to compute all the pairs of instructions that can execute in parallel or only a the most important, the MHP analysis supports other important settings, as shown in the next figure:



Using these parameters we can adapt the behavior of the analysis:

- Debug information (0, 1, or 2): level of information that is printed in the console, where 0 is the lowest level.
- Points-to analysis precision (1, 2, 3 and 4): the precision that will be used in the point-to analysis that approximates the different objects that are created during execution.
- Condition synchronization MHP extension (no, yes): by default, the MHP analysis only considers simple await instructions involving one future variable (await f?). Enabling this option the analysis tries to use the information from await instructions that involve complex conditions like await x != null.
- Inter-Procedural Synchronization (no, yes): enables a refinement of the MHP analysis that obtains finer information about the finished task, thus producing more precise results in programs with inter-procedural synchronization.

2.3 May-Happen-in-Parallel with Inter-Procedural Synchronization

Let us analyze with SACO the program `InterProcedural.abs` with the Inter-Procedural Synchronization of the MHP analysis (described in this this paper), as mentioned before.

```
1  module Parallel;
2  import * from ABS.StdLib;
3
4  interface IO1 {
5          Unit f ();
6  }
7
8  interface IO2 {
9          Unit g (Fut<Unit> w);
10 }
11
12 class O1 implements IO1 {
13         Unit f () {
14                 skip;
15         }
16 }
17
18 class O2 implements IO2 {
19         Unit g (Fut<Unit> w) {
20                 skip;
21                 await w?;
```

```
22                  skip;
23          }
24  }
25
26  {
27                  Fut<Unit> x;
28                  Fut<Unit> y;
29                  IO1 o1 = new O1();
30                  IO2 o2 = new O2();
31                  x = o1!f();
32                  y = o2!g(x);
33                  await y?;
34  }
```

To do so, we select the May-Happen-in-Parallel analysis and set the option Inter-Procedural Synchronization to yes. If we apply the MHP analysis of Section 2.2 we obtain that $L22$ can happen in parallel with $L34$ because it is not able to infer that when method g finishes its execution, method f has finished too (caused by the await instruction in $L21$). Using this refinement we obtain that the only program points that can happen in parallel with $L34$ are the end of methods f and g, which means that both methods must have finished when $L34$ is reached.