The ABC User's Guide

Sébastien BRIAIS

March 4, 2004

Abstract

We describe in this document how to use our tool called Another Bisimilarity Checker (ABC). The ABC is a tool that checks open equivalence between terms of π-calculus.

1 Input syntax: π-calculus terms

1.1 The π-calculus

We assume a countably infinite set of names denoted by $a, b, x, v, \ldots$ and a disjoint set of agent identifiers denoted by X. The syntax of π-calculus terms is described by the following grammar:

$$\begin{array}{rcl} P, Q & ::= & \mathbf{0} \mid P|Q \mid P+Q \mid \alpha.P \mid (\nu\tilde{x})P \mid \mathrm{X}(\tilde{v}) \mid [a=b]P \\ \alpha & ::= & a(\tilde{x}) \mid \overline{a}\langle\tilde{v}\rangle \mid \tau \end{array}$$

As usual, we write $\tilde{v}$ as a shortcut for $v_1, \ldots, v_n$ for some n.

Sometimes, π-calculus syntax is presented in terms of concretions and abstractions. A concretion is mainly something like $[\tilde{v}]C$ where C is a process or a concretion. An abstraction is mainly something like $(\lambda\tilde{x})A$ where A is a process or an abstraction. In this setting, prefixes are either τ, an input name a or an output name $\overline{a}$. The presentation in terms of concretions and abstractions make the labelled-transition semantics easier to write. It is easy to convert classic syntax to this one. Indeed, we have

$$\begin{array}{rcl} a(\tilde{x}).P & \equiv & a.(\lambda\tilde{x})P \\ \overline{a}\langle\tilde{v}\rangle.P & \equiv & \overline{a}.[\tilde{v}]P \end{array}$$

The π-calculus terms accepted by the ABC are the one described by the previous grammar. It also allows the user to enter a π-calculus term written in terms of abstractions and concretions.

The ABC also allows to enter a term partially applied to a variable. This makes sense only if the term is an abstraction.

1.2 Input syntax

```
agent    ::=  factor
           |  factor "+" agent

factor   ::=  term
           |  term "|" factor

term     ::=  "0"
           |  ident params
           |  prefix "." term
           |  "[" vars "]" term
           |  "[" var "=" var "]" term
           |  "(" "\" vars ")" term
           |  "(" "^" vars ")" term
           |  term var
           |  "(" agent ")

vars     ::=  var
           |  var "," vars

prefix   ::=  "t"
           |  var params
           |  "'" var oparams

params   ::=  ε
           |  "(" vars ")"

oparams  ::=  ε
           |  "<" vars ">"

var      ::=  [a-z][a-zA-Z_/.0-9]*

ident    ::=  [A-Z][a-zA-Z_/.0-9]*
```

Remark: Actually, it is optional to write the nil process after an action.

2 Commands of the ABC

2.1 Agent definition

Syntax
agent ident params = agent
Note that only *closed agents*[1] are allowed (i.e. fn(agent) ⊂params).

2.2 Checking (open) equivalence

2.2.1 Strong equivalence

Syntax
eq agent1 agent2
eqd (vars) agent1 agent2
The command **eq** checks strong equivalence between agent1 and agent2. The command **eqd** takes into account the distinction such that each variable in vars is distinct from all other free names in agent1 and agent2.

[1] for efficiency reasons, see also subsection 2.6

2.2.2 Weak equivalence

Syntax
weq agent1 agent2
weqd (vars) agent1 agent2

Similar to strong equivalence.

2.3 Viewing agents

Syntax
print agent
show agent

The command **print** pretty-prints the given agent whereas the command **show** pretty-prints the standard form of the given agent.

2.4 Exploring the behaviour of an agent

Syntax
step agent

The command **step** enters in an interactive mode in which it is possible to view and simulate the (strong) commitments of the given agent.

2.5 Setting memory usage

Syntax
scale integer
value integer

The command **scale** fixes the scale for setting the memory usage. The command **value** sets the memory usage to $\frac{\textbf{value}}{\textbf{scale}}$. This rational number represents the frequency for caching computed data.

2.6 Implicit context

We found annoying to not allow open terms in agent definition (as mentioned in subsection 2.1). So, we added the notion of semi-explicit context that we call implicit context. The idea is that a context is a list of variables that are implicit in agent definition and agent application. More formally, there is a *current implicit context*, and each agent has an *agent implicit context*. If an agent is not defined, then its implicit context is the current one. So, when an agent is defined, its context is the current one. It is possible to push or pop variables in the current implicit context. When one pops variables out of the current context, these variables are also popped out of all the agent implicit contexts.

Syntax
push var_1 ... var_n
This command pushes var_1 ... var_n into the current context.

pop $[n]$
This command pops n variables out of the current context (n is a positive integer, if n is not specified, then 1 is the value by default).

implicit
This command shows the current implicit context.

clear ident_1 ... ident_n
This command undefines the agent ident_1 ... ident_n if they were defined.

2.7 Other commands

2.7.1 Help

Syntax
help
The command **help** give the user some help.

2.7.2 Resetting the ABC

Syntax
reset
This command clears all the memory of the ABC.

2.7.3 Loading a file

Syntax
load filename
This command loads the given file in the ABC.
$\mathsf{filename} \in \mathsf{var} \cup \mathsf{ident}$.

2.7.4 Exiting

Syntax
exit
exit makes you exit the ABC.

3 Examples

3.1 A beginner's example

Let us start the ABC.

```
bash-2.05b$ ./abc.opt
Welcome to Another Bisimulation Checker
```

Let us define agents A and B with

$$\begin{array}{rcl} \mathrm{A}(x,y) & \stackrel{\mathrm{def}}{=} & x.\mathbf{0}|\overline{y}.\mathbf{0} \\ \mathrm{B}(x,y) & \stackrel{\mathrm{def}}{=} & x.\overline{y}.\mathbf{0}+\overline{y}.x.\mathbf{0} \end{array}$$

```
abc > agent A(x,y) = x.0 | 'y.0
Agent A is defined.
abc > agent B(x,y) = x.'y.0 + 'y.x.0
Agent B is defined.
```

It is well known that $\mathrm{A}(x,y) \not\sim^{o}_{\emptyset} \mathrm{B}(x,y)$[2].

```
abc > eq A(x,y) B(x,y)
The two agents are not strongly related (2).
Do you want to see a trace of execution (yes/no) ? yes
trace of A x y
 -t-> 0
trace of B x y
```

Indeed, $\mathrm{A}(x,y)$ can perform a τ commitment that $\mathrm{B}(x,y)$ can't (as shown in the trace).

```
abc > step A(x,y)
1: { x=y }  =>  A x y --t-> 0

2: {  }  =>  A x y --x-> 'y.0

3: {  }  =>  A x y --'y-> x.0

Please choose a commitment (between 1 and 3) or 0 to exit: 1
no more commitments
abc > step B(x,y)
1: {  }  =>  B x y --x-> 'y.0

2: {  }  =>  B x y --'y-> x.0

Please choose a commitment (between 1 and 2) or 0 to exit: 1
1: {  }  =>  'y.0 --'y-> 0

Please choose a commitment (between 1 and 1) or 0 to exit: 1
no more commitments
```

[2] $\sim^{o}_{D}$ is the open bisimilarity with the set of distinctions D.

Remark: We see above that the commitments are shown numbered. Each commitment has the form:

```
a set of conditions => agent --action--> agent
```

The set of conditions is the one that should be satisfied in order for the commitment to take place.

Thus, it is easy to see that $\mathrm{A}(x, y) \sim^{\mathrm{o}}_{\{x \neq y\}} \mathrm{B}(x, y)$.

```
abc > eqd (x,y) A(x,y) B(x,y)
The two agents are strongly related (3).
Do you want to see the core of the bisimulation (yes/no) ? yes
{
  (
    x.0
    {
      (x, y)
    }
    x.0
  )

  (
    'y.0
    {
      (x, y)
    }
    'y.0
  )

  (
    ('y.x.0 + x.'y.0)
    {
      (x, y)
    }
    ('y.0 | x.0)
  )
}
```

Remark: What we call the *core* of a bisimulation is a ternary relation between an agent, a set of distinctions and an other agent such that its symmetric closure plus the identity relation is a bisimulation.

Moreover, if we define an agent C such that

$$\mathrm{C}(x, y) \quad \stackrel{\mathrm{def}}{=} \quad \mathrm{B}(x, y) + [x = y]\tau.\mathbf{0}$$

```
abc > agent C(x,y) = B(x,y) + [x=y]t.0
Agent C is defined.
```

We have $\mathrm{A}(x, y) \sim_{\emptyset}^{o} \mathrm{C}(x, y)$.

```
abc > eq A(x,y) C(x,y)
The two agents are strongly related (4).
Do you want to see the core of the bisimulation (yes/no) ? yes
{
  (
    0
    { }
    0
  )

  (
    x.0
    { }
    x.0
  )

  (
    'y.0
    { }
    'y.0
  )

  (
    ([x=y]t.0 + 'y.x.0 + x.'y.0)
    { }
    ('y.0 | x.0)
  )
}
```

End of this example.

```
abc > exit
bash-2.05b$
```

3.2 Usage of the implicit context

3.2.1 Basic example

At the beginning, the current implicit context is empty.

```
bash-2.05b$ ./abc
Welcome to Another Bisimulation Checker
abc > implicit
No implicit variables.
```

Assume that we want to define an agent $\mathrm{A}(x) \stackrel{\text{def}}{=} x.\mathbf{0} + \tau.\mathrm{A}(x)$, but with using the concept of implicit context.

We first push x into the current context.

```
abc > push x
Pushing x
abc > implicit
Implicit variables are x
```

We then define our agent A.

```
abc > agent A = x.0 + t.A
Agent A is defined.
```

Let us see what is the standard form of A.

```
abc > show A
( x. 0 +  t. A x )
```

Note that when we write A, it really means $A\ x$, since the implicit context of A is x.

Now, we push another name y, see the standard form of A and then we pop the two names we had pushed and see again the standard form of A.

```
abc > push y
Pushing y
abc > implicit
Implicit variables are x y
abc > show A
( x. 0 +  t. A x )
abc > pop 2
Popping at most 2 variables.
abc > implicit
No implicit variables.
abc > show A
(\x0) ( x0. 0 +  t. A x0 )
abc > exit
bash-2.05b$
```

3.2.2 Advanced example

One difficulty that appears with the concept of implicit context is the behavior when there is mutually recursive agent definitions.

In the following, we define two mutually recursive agents A and B, with $\mathrm{A}(x) = x(y).\mathrm{B}(x, y)$ and $\mathrm{B}(x, y) = \overline{x}.\mathrm{A}(x)$.

```
bash-2.05b$ ./abc
Welcome to Another Bisimulation Checker
abc > push x
```

```
Pushing x
abc > agent A = x(y).B y
Agent A is defined.
abc > show A
x(x0). B x x0
abc > push y
Pushing y
abc > agent B = 'x.A
Agent B is defined.
abc > show B
'x. A x
abc > pop
Popping at most 1 variables.
abc > implicit
Implicit variables are x
abc > show A
x(x0). B x x0
abc > show B
(\x0) 'x. A x
abc > show A
(\x0) x0(x1). B x0 x1
abc > show B
(\x0, x1) 'x0. A x0
```

Now, we redo the same actions, to illustrate the use of **clear**.

```
abc > push x
Pushing x
abc > agent A = x(y).B y
Agent A is defined.
abc > show A
x(x0). B x0
```

We see here that the behaviour is not the one we want. Indeed, now x is not an implicit parameter of B since B is already defined and the implicit context of B is now empty. So, we have to clear the definition of B before redoing the same commands.

```
abc > clear B
Clearing B
abc > agent A = x(y).B y
Agent A is defined.
abc > show A
x(x0). B x x0
```

And then, we can continue as before.