The ABC User's Guide

Sébastien BRIAIS

May 10, 2005

Abstract

In this document, we describe how to use the ABC (Another Bisimilarity Checker) which is a tool that checks for open equivalence between terms of the π-calculus. Open bisimulation was first defined by Sangiorgi [San96]. The ABC implements the equivalence checker of the Mobility Workbench [Vic94]. In the following, we assume the reader is familiar with the π-calculus.

The ABC is released under the GPL. The distribution can be downloaded from the ABC homepage located at:

`http://lampwww.epfl.ch/~sbriais/abc.html`

1 Input syntax: π-calculus terms

1.1 The π-calculus of ABC

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 $[\tilde{v}]C$ where C is a process or a concretion. An abstraction is $(\lambda\tilde{x})A$ 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.

Remark: We sometimes use the word *name* and sometimes the word *variable*. However, these two words designate the same concept because in the case of open bisimulation, the two notions coincides.

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: It is optional to write the nil process after an action.

2 Commands of the ABC

2.1 Agent definition

agent ident params = agent

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

2.2 Checking (open) equivalence

An open bisimulation is a family of binary relations between processes of the π-calculus that is indexed by a set of *distinctions*. A distinction D is an irreflexive and symmetric binary relation between names. It represents all the disequalities that should hold. It can also be used to "simulate" free constant names. For instance, assuming that you have defined an agent $P(a, b, c)$ and that you want a to be a constant name distinct from each other name, you should work with the initial distinction $\{(a, b), (a, c), (b, a), (c, a)\}$.

[1] for efficiency reasons, see also subsection 3.2

2.2.1 Strong bisimilarity

eq agent1 agent2
eqd (vars) agent1 agent2

The command **eq** checks strong open bisimilarity 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.

2.2.2 Weak bisimilarity

weq agent1 agent2
weqd (vars) agent1 agent2

Usage similar to strong bisimilarity.

2.2.3 Strong similarity

lt agent1 agent2
ltd (vars) agent1 agent2

Usage similar to strong bisimilarity.

2.2.4 Weak similarity

wlt agent1 agent2
wltd (vars) agent1 agent2

Usage similar to strong bisimilarity.

2.3 Viewing agents

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.

Recall that every abstraction F can be converted to a standard form $F \equiv (\lambda\tilde{x})P$ by pushing restrictions inwards and that every concretion C can be converted to a standard form $C \equiv (\nu\tilde{y})[\tilde{x}]P$ with $\tilde{y} \subset \tilde{x}$ by pulling restrictions on names in $\tilde{x}$ outwards and pushing all others inwards.

2.4 Exploring the behaviour of an agent

step agent

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

2.5 Setting memory usage

scale integer
value integer

By default, ABC stores all the computations. However, this can be too much memory consuming for large systems. So, it is possible to tell ABC to store only a fraction of the computations defined by the ratio $\frac{\textbf{value}}{\textbf{scale}}$.

2.6 Other commands

2.6.1 Help

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

2.6.2 Resetting the ABC

reset
This command clears all the memory of the ABC.

2.6.3 Loading a file

load "filename"
This command loads the given file in the ABC.

2.6.4 Exiting

exit
exit makes you exit the ABC.

3 Examples

3.1 A beginner's example

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

After having started the ABC, we define the 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 $\mathrm{A}(x,y) \not\sim^{\mathrm{o}}_{\emptyset} \mathrm{B}(x,y)$[2].

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

```
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
```

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, $\mathrm{A}(x, y) \sim^{\circ}_{\{x \neq y\}} \mathrm{B}(x, y)$.

```
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

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

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

We have $\mathrm{A}(x,y) \sim^{\circ}_{\emptyset} \mathrm{C}(x,y)$.

```
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 Implicit context

What about free names? The notion of *implicit context* is our answer to the limitation explained in Subsection 2.1. It appears often, when defining a system, that several agents use exactly the same parameters that can be seen as free names on which the system depends. The following example illustrates this fact:

$$\begin{aligned} \mathrm{Buf}_{20}(in, out) &= in(x).\mathrm{Buf}_{21}(in, out, x) \\ \mathrm{Buf}_{21}(in, out, x) &= in(y).\mathrm{Buf}_{22}(in, out, x, y) + \overline{out}\langle x\rangle.\mathrm{Buf}_{20}(in, out) \\ \mathrm{Buf}_{22}(in, out, x, y) &= \overline{out}\langle y\rangle.\mathrm{Buf}_{21}(in, out, x) \end{aligned}$$

If free names were allowed, we would have simply assumed to have two names in and out. Then, we could have written:

$$\begin{aligned} \mathrm{Buf}_{20} &= in(x).\mathrm{Buf}_{21}(x) \\ \mathrm{Buf}_{21}(x) &= in(y).\mathrm{Buf}_{22}(x, y) + \overline{out}\langle x\rangle.\mathrm{Buf}_{20} \\ \mathrm{Buf}_{22}(x, y) &= \overline{out}\langle y\rangle.\mathrm{Buf}_{21}(x) \end{aligned}$$

This mechanism is precisely the one behind implicit contexts (actually, it should be called semi-explicit context).

The idea is that a context is a stack of names. Each agent has its own context. Moreover, there is a *current context* which is global. When an agent is not defined, its context is the current one (which evolves dynamically). It is possible to push names into the current context or to push names out of it. When an agent is defined, its context at the moment of its definition is the

current one and it is duplicated to be its own context. The context of an agent can only shrink but not grow. So, when a name is pushed into the current context, the contexts of the defined agents do not evolve whereas when a name is popped out of the current context, this name is popped out of the contexts of all the defined agents that contained it. When using the implicit contexts, it is a good idea to have in mind the following scheme:

```
(* A and B are undefined *)
(* the current context is empty *)
push x
  (* the context of A is x *)
  (* the context of B is x *)
  agent A = ... B ... (* the context of A is x *)
  push y
    (* the context of A is x *)
    (* the context of B is x y *)
    agent B =  ... (* the context of B is x y *)
    push z
      (* the context of A is x *)
      (* the context of B is x y *)
    ...
    pop
    (* the context of A is x *)
    (* the context of B is x y *)
  pop
  (* the context of A is x *)
  (* the context of B is x *)
pop
(* the context of A is empty *)
(* the context of B is empty *)
push x
  (* the context of A is empty *)
  (* the context of B is empty *)
  agent A = ... B ... (* the context of A is x *)
  ...
pop
(* the context of A is empty *)
(* the context of B is empty *)
```

When an agent A is used in the definition of another agent B, the agent A is partially applied to its context. So, if the context of A was x, y and the one of B was x, y, z, writing A into the definition of B is understood as $A(x, y)$ (in fact: $A\ x\ y$). We warn the reader to be very careful when, for example, re-loading a file (or redefining an agent). Indeed, since the context of an undefined agent is the current one and the context of an already defined agent is its own one, there may be unintended effects when re-defining two (or more) mutually recursive agents. That is why there is a command to undefine an agent.

Here are the added commands to handle implicit contexts:

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.
The next subsection rewrites the example of this subsection in ABC.

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.
```

We first push *in* and *out* into the current context.

```
abc > push in out
Pushing in out
abc > implicit
Implicit variables are in out
```

We then define the agent Buf_{20}.

```
abc > agent Buf20 = in(x).Buf21(x)
Agent Buf20 is defined.
```

Let us check the standard form of Buf_{20}.

```
abc > show Buf20
in(x0). Buf21 in out x0
```

Note that when the user write Buf_{21}, it really means Buf_{21} *in out*, since the implicit context of Buf_{21} is the current one which is *in out* as shown by the command `implicit`.

```
abc > implicit
Implicit variables are in out
```

We continue by defining the agents Buf_{21} and Buf_{22}. However, we can notice that x is also a common additionnal parameter of these two agents. So, we add x to the current context and then define the agents.

```
abc > push x
Pushing x
abc > agent Buf21 = 'out<x>.Buf20 + in(y).Buf22 y
Agent Buf21 is defined.
abc > agent Buf22(y) = 'out<y>.Buf21
Agent Buf22 is defined.
```

We then display the standard form of the three agents and the current context.

```
abc > implicit
Implicit variables are in out x
abc > show Buf20
in(x0). Buf21 in out x0
abc > show Buf21
( 'out<x>. Buf20 in out +  in(x0). Buf22 in out x x0 )
abc > show Buf22
(\x0) 'out<x0>. Buf21 in out x
```

We can now pop all the "free names" and redo the same actions.

```
abc > pop 3
Popping x out in
abc > implicit
No implicit variables.
abc > show Buf20
(\x0, x1) x0(x2). Buf21 x0 x1 x2
abc > show Buf21
(\x0, x1, x2) ( 'x1<x2>. Buf20 x0 x1 +  x0(x3). Buf22 x0 x1 x2 x3 )
abc > show Buf22
(\x0, x1, x2, x3) 'x1<x3>. Buf21 x0 x1 x2
```

If we prefer, in a more readable setting:

```
abc > show Buf20 in out
in(x0). Buf21 in out x0
abc > show Buf21 in out x
( 'out<x>. Buf20 in out +  in(x0). Buf22 in out x x0 )
abc > show Buf22 in out x y
'out<y>. Buf21 in out x
```

References

[San96] Davide Sangiorgi. A theory of bisimulation for the π-calculus. *Acta Informatica*, 33:69–97, 1996. Earlier version published as Report ECS-

LFCS-93-270, University of Edinburgh. An extended abstract appeared in the *Proceedings of CONCUR '93*, LNCS 715.

[Vic94] Björn Victor. *A Verification Tool for the Polyadic π-Calculus*. Licentiate thesis, Department of Computer Systems, Uppsala University, Sweden, May © 1994. Available as report DoCS 94/50.