Tutorial on Specification and Verification of VERDS Models

Wenhui Zhang
State Key Laboratory of Computer Science, Institute of Software, Chinese Academy of Sciences
P.O.Box 8718, Beijing 100190, China

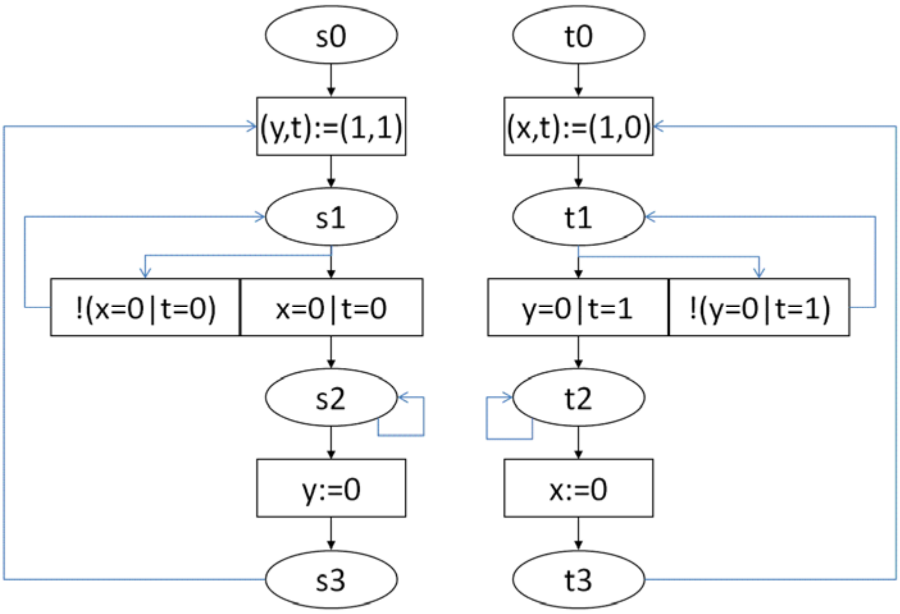
2012-12-19

This document contains a tutorial on specification in *verds* models (or *verds* verification models, VVMs for short) and on running the verification tool *verds*.

1. Mutual Exclusion

This part of the tutorial uses the mutual exclusion algorithm to explain how to use the verification tools *verds* including how to specify the verification models.

The following figure shows a flow diagram of a simple mutual exclusion algorithm with two processes.



Let A denote the process on the left part of the figure and B denote the right one. The state s0 represents the initial state of A. The state s1 represents the requesting state of A. The state s2 represents that A is at the critical state that allows A to use the exclusively shared resource. The state s3 represents that A is at a non-critical state. The interpretation of t0, t1, t2 and t3 is similar for the process B.

The possible state transitions of process A are as follows:

(1)at state s0: A may set (y,t) to (1,1) and proceed to state s1;

(2)at state s1: A may proceed to state s2 when (x=0|t=0) holds, or check and go back to state s1 when the condition does not hold;

(3)at state s2: A may set y to 0 and proceed to state s3, or do some unspecified local action and stay at state s2;

(4)at state s3: A may set (y,t) to (1,1) and proceed to state s1;

The possible state transitions of process B are similar.

Let a be a variable with values in {s0,s1,s2,s3} that keeps the local state of A, and b be a variable with values in {t0,t1,t2,t3} that keeps the local state of B. The initial state of the algorithm is as follows:

a=s0
b=t0
x=0
y=0
t=0

The desired properties of the algorithm include the following ones.

(1)mutual exclusion: !(a=s2&b=t2)

(2)progress: (a=s1->AF(a=s2|b=t2))&(b=t1->AF(a=s2|b=t2))

(3)non-starvation: (a=s1->AF(a=s2))&(b=t1->AF(b=t2))

(4)cooperative non-starvation: (a=s1->EF(a=s2))&(b=t1->EF(b=t2))

For the generality of the specification, the above formulas are to be preceded with the temporal operator AG.

1.1. Specification of Mutual Exclusion in VVM

We explain how the mutual exclusion algorithm may be specified in VVM in several different ways.

1.1.1. Specification with a Single Process

The strait forward way of specification is to put all transitions of the processes together to make a single process. Then we have the following specification of the mutual exclusion algorithm.

```
VVM     me001
VAR
        x: 0..1;
        y: 0..1;
        t: 0..1;
        a: {s0,s1,s2,s3};
        b: {t0,t1,t2,t3};
INIT
        x=0;
        y=0;
        t=0;
        a=s0;
        b=t0;
TRANS
        a=s0:               (y,t,a):=(1,1,s1);
        a=s1&(x=0|t=0):     (a):=(s2);
        a=s1&!(x=0|t=0):    (a):=(s1);
        a=s2:               (y,a):=(0,s3);
        a=s2:               (a):=(s2);
        a=s3:               (y,t,a):=(1,1,s1);
        b=t0:               (x,t,b):=(1,0,t1);
        b=t1&(y=0|t=1):     (b):=(t2);
        b=t1&!(y=0|t=1):    (b):=(t1);
        b=t2:               (x,b):=(0,t3);
        b=t2:               (b):=(t2);
        b=t3:               (x,t,b):=(1,0,t1);
SPEC
        AG(!(a=s2&b=t2));
        AG((!a=s1|AF(a=s2|b=t2))&(!b=t1|AF(a=s2|b=t2)));
        AG((!a=s1|AF(a=s2))&(!b=t1|AF(b=t2)));
        AG((!a=s1|EF(a=s2))&(!b=t1|EF(b=t2)));
```

1.1.2. Specification with Two Modules

We may separate the specification of the processes by defining two different modules, one for each process. Then we have the following specification of the mutual exclusion algorithm.

```
VVM     me002
VAR
        x: 0..1;
        y: 0..1;
        t: 0..1;
INIT
        x=0;
        y=0;
        t=0;
PROC
        p0: p0m();
        p1: p1m();
SPEC
        AG(!(p0.a=s2&p1.b=t2));
        AG((!p0.a=s1|AF(p0.a=s2|p1.b=t2))&(!p1.b=t1|AF(p0.a=s2|p1.b=t2)));
        AG((!p0.a=s1|AF(p0.a=s2))&(!p1.b=t1|AF(p1.b=t2)));
        AG((!p0.a=s1|EF(p0.a=s2))&(!p1.b=t1|EF(p1.b=t2)));

MODULE  p0m()
VAR
        a: {s0,s1,s2,s3};
INIT
        a=s0;
TRANS
        a=s0:            (y,t,a):=(1,1,s1);
        a=s1&(x=0|t=0):  (a):=(s2);
        a=s1&!(x=0|t=0):(a):=(s1);
        a=s2:            (y,a):=(0,s3);
        a=s2:            (a):=(s2);
        a=s3:            (y,t,a):=(1,1,s1);

MODULE  p1m()
VAR
        b: {t0,t1,t2,t3};
INIT
        b=t0;
```

```
TRANS
        b=t0:             (x,t,b):=(1,0,t1);
        b=t1&(y=0|t=1):   (b):=(t2);
        b=t1&!(y=0|t=1):  (b):=(t1);
        b=t2:             (x,b):=(0,t3);
        b=t2:             (b):=(t2);
        b=t3:             (x,t,b):=(1,0,t1);
```

1.1.3. Specification with One module

We may combine the two modules into one, such that it can be instantiated differently for the two processes. Then we have the following specification of the mutual exclusion algorithm.

```
VVM     me003
VAR
        x: 0..1;
        y: 0..1;
        t: 0..1;
INIT
        x=0;
        y=0;
        t=0;
PROC
        p0: p0m(x,y,t,0);
        p1: p0m(y,x,t,1);
SPEC
        AG(!(p0.a=s2&p1.a=s2));
        AG((!p0.a=s1|AF(p0.a=s2|p1.a=s2))&(!p1.a=s1|AF(p0.a=s2|p1.a=s2)));
        AG((!p0.a=s1|AF(p0.a=s2))&(!p1.a=s1|AF(p1.a=s2)));
        AG((!p0.a=s1|EF(p0.a=s2))&(!p1.a=s1|EF(p1.a=s2)));

MODULE  p0m(x,y,t,i)
VAR
        a: {s0,s1,s2,s3};
INIT
        a=s0;
TRANS
        a=s0:             (y,t,a):=(1,1-i,s1);
        a=s1&(x=0|t=i):   (a):=(s2);
        a=s1&!(x=0|t=i):  (a):=(s1);
        a=s2:             (y,a):=(0,s3);
        a=s2:           (a):=(s2);
        a=s3:           (y,t,a):=(1,1-i,s1);
```

1.1.4. Specification with Array Variables

We may also use array variables in the specification, such that x and y are replaced by x[0] and x[1]. Then we have the following specification of the mutual exclusion algorithm.

```
VVM     me004
VAR
        x[0..1]: 0..1;
        t: 0..1;
INIT
        x[0]=0;
        x[1]=0;
        t=0;
PROC
        p0: p0m(x[],t,0);
        p1: p0m(x[],t,1);
SPEC
        AG(!(p0.a=s2&p1.a=s2));
        AG((!p0.a=s1|AF(p0.a=s2|p1.a=s2))&(!p1.a=s1|AF(p0.a=s2|p1.a=s2)));
        AG((!p0.a=s1|AF(p0.a=s2))&(!p1.a=s1|AF(p1.a=s2)));
        AG((!p0.a=s1|EF(p0.a=s2))&(!p1.a=s1|EF(p1.a=s2)));

MODULE  p0m(x[],t,i)
VAR
        a: {s0,s1,s2,s3};
INIT
        a=s0;
TRANS
        a=s0:                   (x[1-i],t,a):=(1,1-i,s1);
        a=s1&(x[i]=0|t=i):      (a):=(s2);
        a=s1&!(x[i]=0|t=i):     (a):=(s1);
        a=s2:                   (x[1-i],a):=(0,s3);
        a=s2:                   (a):=(s2);
        a=s3:                   (x[1-i],t,a):=(1,1-i,s1);
```

1.2. Verification with *verds*

Assuming that the VVM is contained in the file named “me001.vvm”. To check whether the i-th (starting from 1) property holds, we use the command

```
verds -ck i me001.vvm
```

with a specified value of i. The result of checking the first property is as follows:

```
system_prompt> verds -ck 1 me001.vvm
VVERSION:    verds 1.42 - DEC 2012
FILE:       me001.vvm
PROPERTY:   A G ! ((a = 2 )& (b = 2 ))
bound =  1  time = 0
----------  time = 0
bound =  2  time = 0
----------  time = 0
bound =  3  time = 0
----------  time = 0
bound =  4  time = 0
----------  time = 0
bound =  5  time = 0
----------  time = 0
bound =  6  time = 0
----------  time = 0
CONCLUSION: TRUE (time=0)
```

The first line is the command and the rest is the output of the verification. In the output, the first line tells the version of the verification tool, the second line tells the input file, the third line tells the property been checked (note that in the output, the enumerative constants have been replaced by numerical ones), and the last line is the conclusion and the running time (the clock time in seconds, not the CPU time).

It is similar for checking the other three properties. In summary, the conclusions are as follows.

Property	Conclusion

AG(!(p0.a=2&p1.a=2))	true
AG((!p0.a=1|AF(p0.a=2|p1.a=2))&(!p1.a=1|AF(p0.a=2|p1.a=2)))	false
AG((!p0.a=1|AF(p0.a=2))&(!p1.a=1|AF(p1.a=2)))	false
AG((!p0.a=1|EF(p0.a=2))&(!p1.a=1|EF(p1.a=2)))	true

This model does not satisfy the progress property and the non-starvation property, because one process may keep trying to enter the critical region without success while the other is ready to enter but not trying to make a move into the critical region. For the specification of a refined model, fairness is needed.

1.3. Specification with Fairness

The model may be specified with fairness, in order to force (the valid executions of) a process to make a move. The complete specification of the model is as follows.

```
VVM      me005
VAR
         x[0..1]: 0..1;
         t: 0..1;
INIT
         x[0]=0;
         x[1]=0;
         t=0;
PROC
         p0: p0m(x[],t,0);
         p1: p0m(x[],t,1);
SPEC
         AG(!(p0.a=s2&p1.a=s2));
         AG((!p0.a=s1|AF(p0.a=s2|p1.a=s2))&(!p1.a=s1|AF(p0.a=s2|p1.a=s2)));
         AG((!p0.a=s1|AF(p0.a=s2))&(!p1.a=s1|AF(p1.a=s2)));
         AG((!p0.a=s1|EF(p0.a=s2))&(!p1.a=s1|EF(p1.a=s2)));

MODULE   p0m(x[],t,i)
VAR
         a: {s0,s1,s2,s3};
INIT
         a=s0;
TRANS
         a=s0:                    (x[1-i],t,a):=(1,1-i,s1);
```

```
        a=s1&(x[i]=0|t=i):        (a):=(s2);
        a=s1&!(x[i]=0|t=i):       (a):=(s1);
        a=s2:                     (x[1-i],a):=(0,s3);
        a=s2:                     (a):=(s2);
        a=s3:                     (x[1-i],t,a):=(1,1-i,s1);
FAIRNESS
        running;
```

The keyword *running* specifies a special fairness requirement meaning that the valid execution sequences are restricted to those in which the process having this fairness requirement executes infinitely many times. In another words, this fairness requirement tells that a process must make a move soon or later, otherwise, the execution trace will not be considered as a valid one and whether it satisfies a property is not interesting.

Then the verification results are as follows.

Property	Conclusion
AG(!(p0.a=2&p1.a=2))	true
AG((!p0.a=1|AF(p0.a=2|p1.a=2))&(!p1.a=1|AF(p0.a=2|p1.a=2)))	true
AG((!p0.a=1|AF(p0.a=2))&(!p1.a=1|AF(p1.a=2)))	false
AG((!p0.a=1|EF(p0.a=2))&(!p1.a=1|EF(p1.a=2)))	true

This model still does not satisfy the non-starvation property, because a process that has entered the critical region may keep the resource forever, such that the other process has no chance to use the property.

For avoiding this, we may add a fairness requirement a!=s2 in order to force (the valid executions of) a process to move out of the critical region once in a while. The modified specification of the model is as follows.

```
VVM     me006
VAR
        x[0..1]: 0..1;
        t: 0..1;
INIT
        x[0]=0;
        x[1]=0;
        t=0;
```

```
PROC
        p0: p0m(x[],t,0);
        p1: p0m(x[],t,1);
SPEC
        AG(!(p0.a=s2&p1.a=s2));
        AG((!p0.a=s1|AF(p0.a=s2|p1.a=s2))&(!p1.a=s1|AF(p0.a=s2|p1.a=s2)));
        AG((!p0.a=s0|AF(p0.a=s2))&(!p1.a=s0|AF(p1.a=s2)));
        AG((!p0.a=s1|EF(p0.a=s2))&(!p1.a=s1|EF(p1.a=s2)));

MODULE  p0m(x[],t,i)
VAR
        a: {s0,s1,s2,s3};
INIT
        a=s0;
TRANS
        a=s0:                   (x[1-i],t,a):=(1,1-i,s1);
        a=s1&(x[i]=0|t=i):      (a):=(s2);
        a=s1&!(x[i]=0|t=i):     (a):=(s1);
        a=s2:                   (x[1-i],a):=(0,s3);
        a=s2:                   (a):=(s2);
        a=s3:                   (x[1-i],t,a):=(1,1-i,s1);
FAIRNESS
        running;
        a!=s2;
```

Then the verification results are as follows.

Property	Conclusion
AG(!(p0.a=2&p1.a=2))	true
AG((!p0.a=1|AF(p0.a=2|p1.a=2))&(!p1.a=1|AF(p0.a=2|p1.a=2)))	true
AG((!p0.a=1|AF(p0.a=2))&(!p1.a=1|AF(p1.a=2)))	true
AG((!p0.a=1|EF(p0.a=2))&(!p1.a=1|EF(p1.a=2)))	true

1.4. Verification with the -bcc Option

This option is for the use of verification based on bounded semantics. The verification approach is currently implemented without taking fairness specification into consideration. To check whether the i-th property holds in the model specified in "me001.vvm" with bounded model checking, we may use the command

```
verds －bcc －ck i me001.vvm
```

with a specified value of i. The result of checking the first property is as follows:

```
system_prompt>  verds -bcc -ck 1 me001.vvm
VERSION:    verds 1.42 - DEC 2012
FILE:       me001.vvm
PROPERTY:   A G ! ((a = 2 )& (b = 2 ))
INFO:       applying an internal SAT-solver
bound =  0  time = 1
----------  time = 1
bound =  1  time = 1
----------  time = 1
bound =  2  time = 1
----------  time = 1
bound =  3  time = 1
----------  time = 1
bound =  4  time = 1
----------  time = 1
bound =  5  time = 1
----------  time = 1
bound =  6  time = 1
----------  time = 1
bound =  7  time = 1
----------  time = 1
bound =  8  time = 1
----------  time = 1
bound =  9  time = 1
----------  time = 1
bound = 10  time = 1
CONCLUSION: TRUE (time=1 bound=10)
```

In the output, the two warning messages remind that there is a possibility to specify a qbf-solver or sat-solver (the latter for verification of ACTL properties) in order to increase the efficiency of the verification. Then the progress is reported, and conclusion is presented with the information on the total time and the bound reached in the bounded model checking.

The verification data for the other three properties are presented as follows.

```
system_prompt>  verds -bcc -ck 2 me001.vvm
```

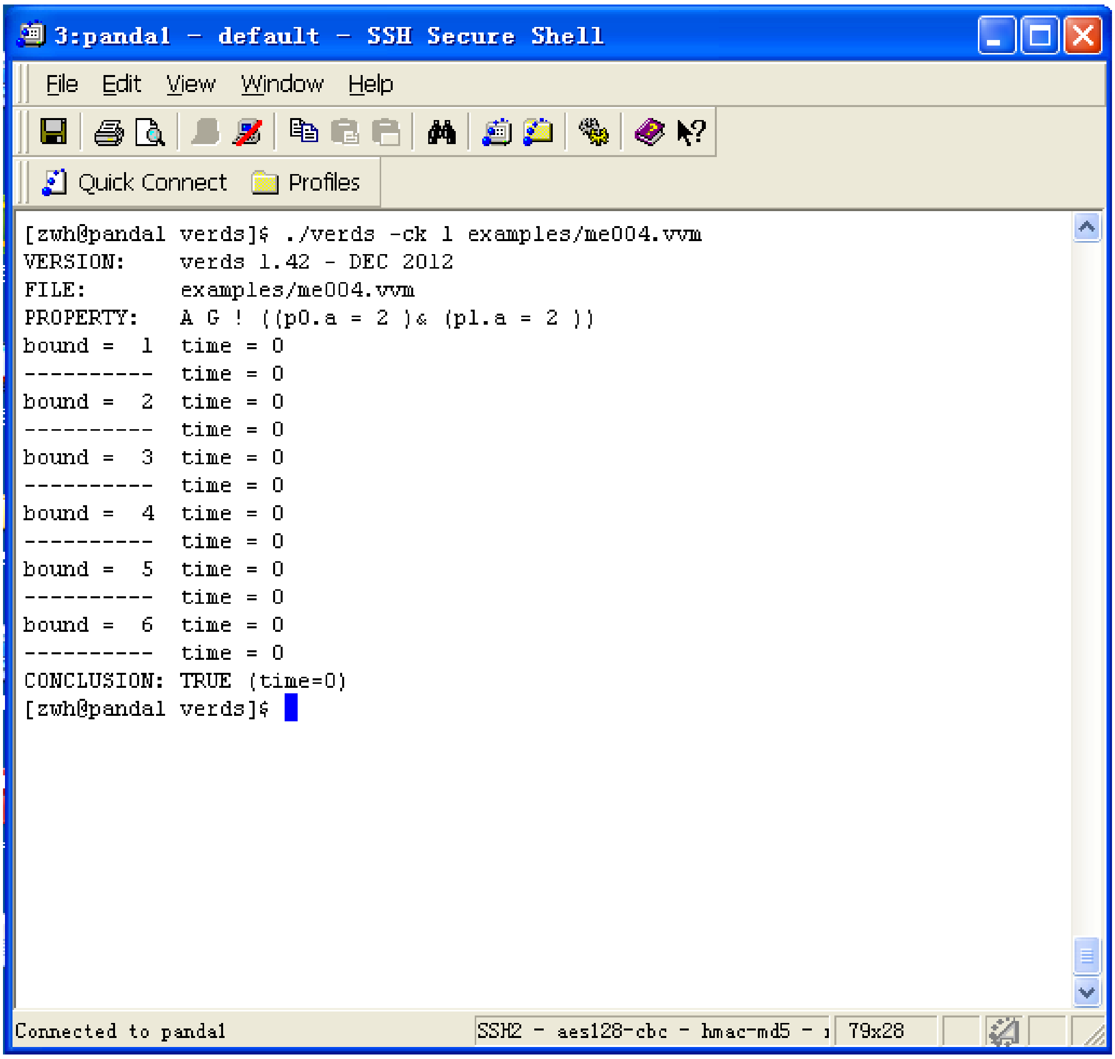
```
VERSION:    verds 1.42 - DEC 2012
FILE:       me001.vvm
PROPERTY:   A G ((! (a = 1 )| A F ((a = 2 )| (b = 2 )))& (! (b = 1 )| A F ((a =
2 )| (b = 2 ))))
INFO:       applying an internal SAT-solver
bound =  0  time = 0
----------  time = 0
bound =  1  time = 0
----------  time = 0
bound =  2  time = 0
----------  time = 0
CONCLUSION: FALSE (time=0 bound=2)

system_prompt>  verds -bcc -ck 3 me001.vvm
VERSION:    verds 1.42 - DEC 2012
FILE:       me001.vvm
PROPERTY:   A G ((! (a = 1 )| A F (a = 2 ))& (! (b = 1 )| A F (b = 2 )))
INFO:       applying an internal SAT-solver
bound =  0  time = 0
----------  time = 0
bound =  1  time = 0
----------  time = 0
bound =  2  time = 0
----------  time = 0
CONCLUSION: FALSE (time=0 bound=2)

system_prompt>  verds -bcc -ck 4 me001.vvm
VERSION:    verds 1.42 - DEC 2012
FILE:       me001.vvm
PROPERTY:   A G ((! (a = 1 )| E F (a = 2 ))& (! (b = 1 )| E F (b = 2 )))
INFO:       applying an internal QBF-solver
bound =  0  time = 0
----------  time = 0
bound =  1  time = 0
----------  time = 0
bound =  2  time = 0
----------  time = 0
bound =  3  time = 0
----------  time = 0
bound =  4  time = 0
----------  time = 0
bound =  5  time = 0
----------  time = 0
bound =  6  time = 0
```

```
----------  time = 2
bound =  7  time = 2
----------  time = 9
bound =  8  time = 11
----------  time = 35
bound =  9  time = 39
----------  time = 124
bound = 10  time = 291
CONCLUSION: TRUE (time=291 bound=10)
```

All properties except the last one are ACTL properties. The complexity of verification of ACTL properties is much lower, since it uses SAT-solving techniques instead of QBF-solving techniques. The efficiency can be enhanced by using more efficient external QBF and SAT solvers.

1.5. Examples of Actual Executions

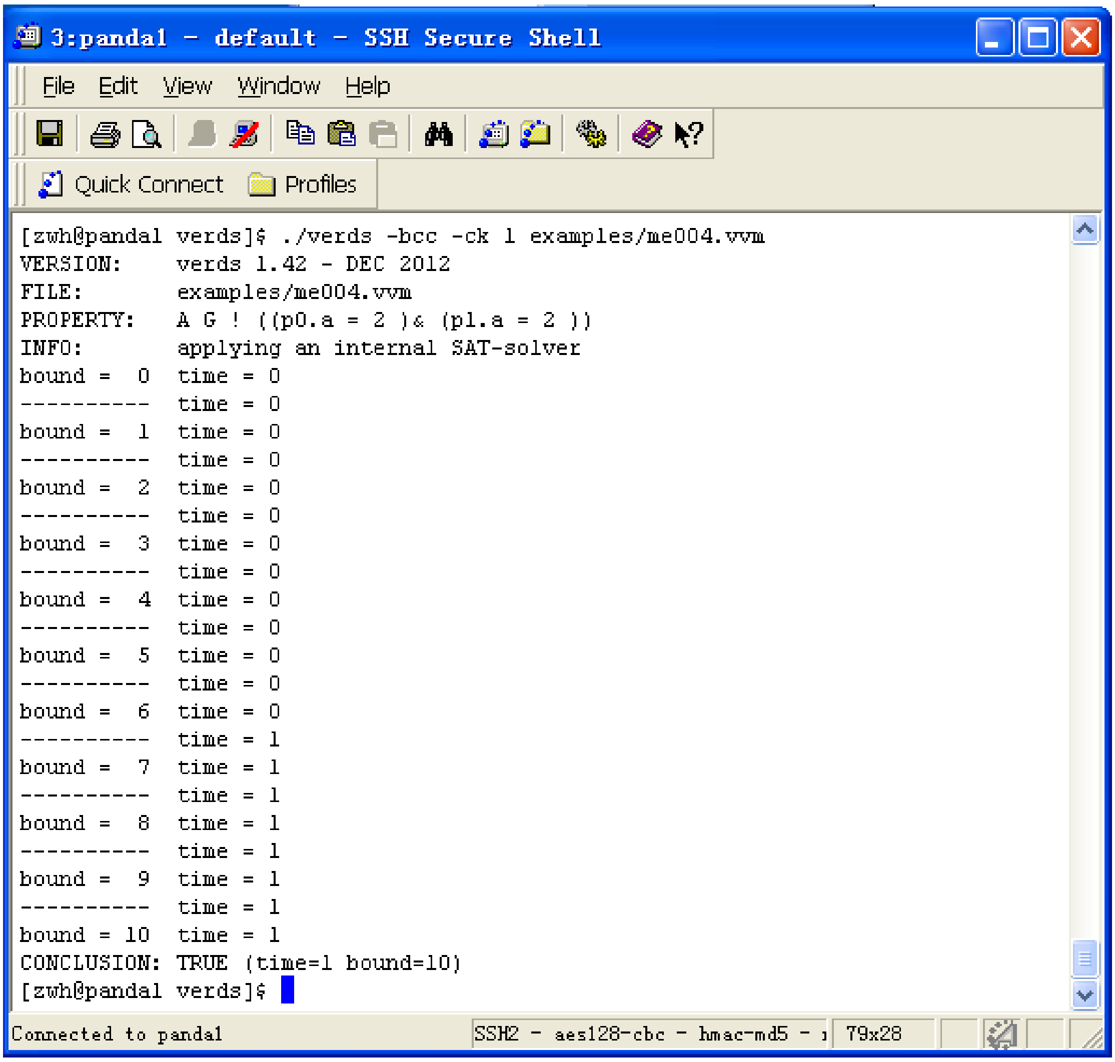
The following is an example of verifying property 1 of the model contained in the file *me004.vvm* located in the subdirectory named *examples*.



3:panda1 - default - SSH Secure Shell
File Edit View Window Help
Quick Connect Profiles
Connected to panda1
SSH2 - aes128-cbc - hmac-md5 - 1
79x28

```
[zwh@panda1 verds]$ ./verds -ck 1 examples/me004.vvm
VERSION:    verds 1.42 - DEC 2012
FILE:       examples/me004.vvm
PROPERTY:   A G ! ((p0.a = 2 )& (p1.a = 2 ))
bound =  1  time = 0
----------  time = 0
bound =  2  time = 0
----------  time = 0
bound =  3  time = 0
----------  time = 0
bound =  4  time = 0
----------  time = 0
bound =  5  time = 0
----------  time = 0
bound =  6  time = 0
----------  time = 0
CONCLUSION: TRUE (time=0)
[zwh@panda1 verds]$
```

The following is an example of verifying the same property with the -*bcc* option.

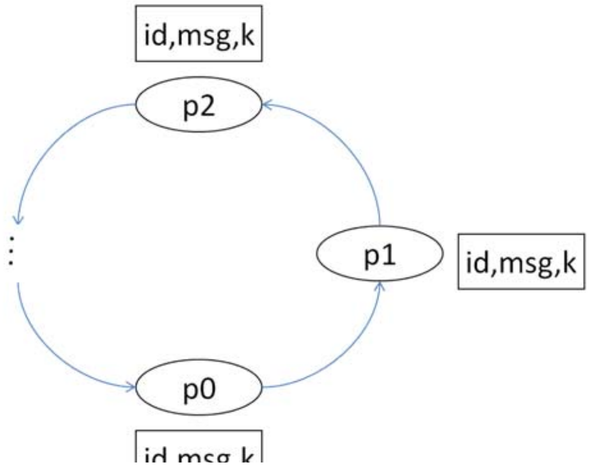

3:panda1 - default - SSH Secure Shell
File
Edit
View
Window
Help
Quick Connect
Profiles
Connected to panda1
SSH2 - aes128-cbc - hmac-md5 -
79x28

```
[zwh@panda1 verds]$ ./verds -bcc -ck 1 examples/me004.vvm
VERSION:     verds 1.42 - DEC 2012
FILE:        examples/me004.vvm
PROPERTY:    A G ! ((p0.a = 2 )& (p1.a = 2 ))
INFO:        applying an internal SAT-solver
bound =  0  time = 0
----------  time = 0
bound =  1  time = 0
----------  time = 0
bound =  2  time = 0
----------  time = 0
bound =  3  time = 0
----------  time = 0
bound =  4  time = 0
----------  time = 0
bound =  5  time = 0
----------  time = 0
bound =  6  time = 0
----------  time = 1
bound =  7  time = 1
----------  time = 1
bound =  8  time = 1
----------  time = 1
bound =  9  time = 1
----------  time = 1
bound = 10  time = 1
CONCLUSION: TRUE (time=1 bound=10)
[zwh@panda1 verds]$
```

2. Leader Election

This part of the tutorial uses the leader election protocol to explain how to use the verification tools *verds* including how to specify the verification models.

To begin with, we have N people sitting in a ring, each with an identification number. Each one is allowed to communicate to the people in the right hand side by sending a message. The state of a person may be characterized by whether he knows (the id of) the leader. An illustration is as follows.



A person acts as follows:

A person is initially in an active state where the leader is not known, and works towards an inactive state where the leader is known to him. In the active state:

(1) In case no messages are received, he may send his id to the next person.

(2) In case a message is received:

a) If the message tells him that a leader (with the id encoded in the message) has already been declared, he records this information and sends the same message to the next person.

b) If the message contains only an id, he may discard it, send it to the next person, or declared that he is the leader and tells the next person this information, according respectively to whether the received id is greater than, less than, or equal to his id.

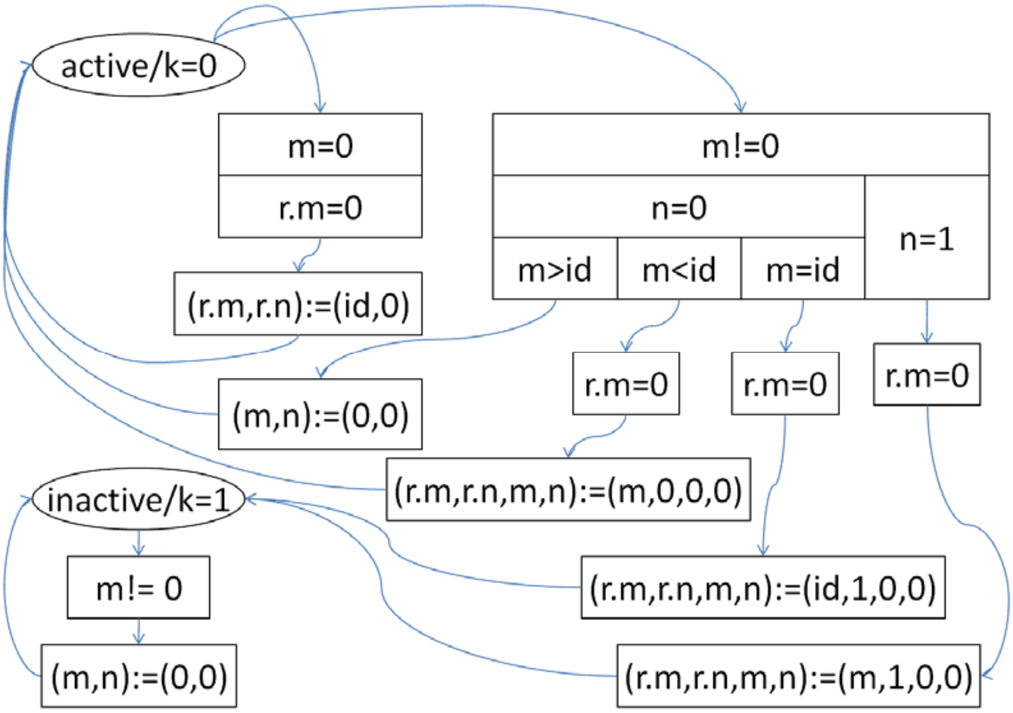
In the inactive state, he may read a message and discard it.

Let id be a variable recording the id of a person. The id is initially a random number in {1,...,N}, different for different person.

Let k be a variable (initially 0) recording that whether the leader is known.

Let m,n be two variables representing the content of the message channel, such that m=0 means that there are no new messages in the message channel, m ∈ {1,..., N} is an id of a person and n ∈ {0,1} indicate whether the leader is not declared or is declared (in the latter case, m is the id of the leader). Let r.m and r.n be the message variables of the right hand side person, and r.m!=0 represent that there is a message in the buffer which has not been read yet.

The process of a person is illustrated as follows for N=3.



Let p0, p1 and p2 denote the three processes. The interesting properties of the protocol include the following ones.

(5) AG(!(p0.k=1) |(AF(p1.n=1)|(p1.k=1)));

(6) AF(p0.k=0 U AG(p0.k=1));

The first property is that if p0 knows the leader, then a message telling that the leader is known must be send to p1, unless p1 also knows the leader; the second is that in all execution paths, p0 will eventually know the leader and remain in this state forever.

2.1. Specification of the Protocol in VVM

We explain how the leader election protocol may be specified in VVM in two different ways.

2.1.1. Specification with the Message Variable as the Parameter

A process may involve two message variables, one his own and the other the message variable of his right hand side process. The latter may be passed as a parameter to the process. Then we have the following specification.

```
VVM       le001
VAR
INIT
          p0.i!=p1.i;
          p0.i!=p2.i;
          p1.i!=p2.i;
PROC
          p0:p0m(p1.m,p1.n);
          p1:p0m(p2.m,p2.n);
          p2:p0m(p0.m,p0.n);
SPEC
          AG(!(p0.k=1)|(AF(p1.n=1)|(p1.k=1)));
          A(p0.k=0 U AG(p0.k=1));

MODULE    p0m(rm,rn)
VAR
          k:0..1;
          i:0..3;
          m:0..3;
          n:0..1;
INIT
          k=0;
          m=0;
          n=0;
          i>=1;
          i<=3;
TRANS
          k=0&m=0&rm=0:             (rm,rn):=(i,0);
          k=0&m!=0&n=0&m>i:         (m,n):=(0,0);
          k=0&m!=0&n=0&m<i&rm=0:    (rm,rn,m,n):=(m,0,0,0);
          k=0&m!=0&n=0&m=i&rm=0:    (rm,rn,m,n,k):=(i,1,0,0,1);
          k=0&m!=0&n=1&rm=0:        (rm,rn,m,n,k):=(m,1,0,0,1);
          k=1&m!=0:                 (m,n):=(0,0);
```

2.1.2. Specification using Array Variables with *pid*

We may an array variable m[0..2], n[0..2] such that m[i], n[i] are the message variable for the i-th process. The keyword pid represents the built-in constant recording the pid of the process using this keyword, and (pid+1)%3 is the pid of the next process. Then we may specify the leader election protocol as follows.

```
VVM     le002
VAR
        m[0..2]:0..3;
        n[0..2]:0..1;
INIT
        p0.i!=p1.i;
        p0.i!=p2.i;
        p1.i!=p2.i;
PROC
        p0:p0m();
        p1:p0m();
        p2:p0m();
SPEC
        AG(!(p0.k=1)|(AF(n[1]=1)|(p1.k=1)));
        A(p0.k=0 U AG(p0.k=1));

MODULE  p0m()
VAR
        k:0..1;
        i:0..3;
INIT
        k=0;
        m[pid]=0;
        n[pid]=0;
        i>=1;
        i<=3;
TRANS
        k=0&m[pid]=0&n[pid]=0&m[(pid+1)%3]=0:
(m[(pid+1)%3],n[(pid+1)%3]):=(i,0);
        k=0&m[pid]!=0&n[pid]=0&m[pid]>i: (m[pid],n[pid]):=(0,0);
        k=0&m[pid]!=0&n[pid]=0&m[pid]<i&m[(pid+1)%3]=0:
(m[(pid+1)%3],n[(pid+1)%3],m[pid],n[pid]):=(m[pid],0,0,0);
        k=0&m[pid]!=0&n[pid]=0&m[pid]=i&m[(pid+1)%3]=0:
(m[(pid+1)%3],n[(pid+1)%3],m[pid],n[pid],k):=(i,1,0,0,1);
        k=0&m[pid]!=0&n[pid]=1&m[(pid+1)%3]=0:
(m[(pid+1)%3],n[(pid+1)%3],m[pid],n[pid],k):=(m[pid],1,0,0,1);
        k=1&m[pid]!=0: (m[pid],n[pid]):=(0,0);
```

2.1.3. Specification with Macros for Readability

We may enhance the readability of the program by defining macros.

Let nid=(pid+1)%3 denote the pid of the next person at the right hand side.

Let ms=(m[nid]=0) denote that one may send a message.

Let mr=(m[pid]!=0) denote that one may read a message (there is some message to read),

Let nk=(n[pid]=0) denote that the leader is not known yet. Then we may specify the leader election protocol as follows with a section that defines nid, ms, mr, and nk.

```
VVM     le003
DEFINE
        nid     =(pid+1)%3
        mr      =(m[pid]!=0)
        ms      =(m[nid]=0)
        nk      =(n[pid]=0)
VAR
        m[0..2]:0..3;
        n[0..2]:0..1;
INIT
        p0.i!=p1.i;
        p0.i!=p2.i;
        p1.i!=p2.i;
PROC
        p0:p0m();
        p1:p0m();
        p2:p0m();
SPEC
        AG(!(p0.k=1)|(AF(n[1]=1)|(p1.k=1)));
        A(p0.k=0 U AG(p0.k=1));

MODULE  p0m()
VAR
        k:0..1;
        i:0..3;
INIT
        k=0;
        m[pid]=0;
        n[pid]=0;
        i>=1;
        i<=3;
TRANS
        k=0&!mr&ms:             (m[nid],n[nid]):=(i,0);
        k=0&mr&nk&m[pid]>i:     (m[pid],n[pid]):=(0,0);
        k=0&mr&nk&m[pid]<i&ms:  (m[nid],n[nid],m[pid],n[pid]):=(m[pid],0,0,0);
        k=0&mr&nk&m[pid]=i&ms:   (m[nid],n[nid],m[pid],n[pid],k):=(i,1,0,0,1);
        k=0&mr&!nk&ms:      (m[nid],n[nid],m[pid],n[pid],k):=(m[pid],1,0,0,1);
        k=1&mr:                  (m[pid],n[pid]):=(0,0);
```

2.2. Verification with *verds*

Assuming that the VVM is contained in the file named “le001.vvm”. To check whether the i-th property holds in the model, we use the command

```
verds  -ck i le001.vvm
```

with a specified value of i. The result of checking the first property is as follows:

```
system_prompt> verds  -ck 1 le001.vvm
VERSION:    verds 1.42 - DEC 2012
FILE:       le001.vvm
PROPERTY:   A G (! (p0.k = 1 )| (A F (p1.n = 1 )| (p1.k = 1 )))
bound =  1  time = 0
----------  time = 0
bound =  2  time = 0
----------  time = 0
bound =  3  time = 0
----------  time = 0
bound =  4  time = 0
----------  time = 0
bound =  5  time = 0
----------  time = 0
bound =  6  time = 0
----------  time = 0
bound =  7  time = 0
----------  time = 0
bound =  8  time = 0
----------  time = 0
bound =  9  time = 0
----------  time = 0
bound = 10  time = 0
----------  time = 0
CONCLUSION: TRUE (time=0)
```

It is similar for checking A(p0.k=0 U AG(p0.k=1)). In summary, the conclusions are as follows.

Property	Conclusion
AG(!(p0.k=1)|(AF(p1.n=1)|(p1.k=1)))	true
A(p0.k=0 U AG(p0.k=1))	false

This model does not satisfy the second property, because not all processes are obligated to send messages to the next one. A fairness condition may be added to exclude some sequence of actions as invalid ones.

2.3. Specification with Fairness

The model may be specified with fairness, in order to force (the valid executions of) a process to send a message unless it is in the state of inactiveness (k=1). The specification is as follows.

```
VVM     le004
DEFINE
        nid    =(pid+1)%3
        mr      =(m[pid]!=0)
        ms      =(m[nid]=0)
        nk      =(n[pid]=0)
VAR
        m[0..2]:0..3;
        n[0..2]:0..1;
INIT
        p0.i!=p1.i;
        p0.i!=p2.i;
        p1.i!=p2.i;
PROC
        p0:p0m();
        p1:p0m();
        p2:p0m();
SPEC
        AG(!(p0.k=1)|(AF(n[1]=1)|(p1.k=1)));
        A(p0.k=0 U AG(p0.k=1));

MODULE  p0m()
VAR
        k:0..1;
        i:0..3;
INIT
        k=0;
        m[pid]=0;
        n[pid]=0;
        i>=1;
        i<=3;
TRANS
        k=0&!mr&ms:              (m[nid],n[nid]):=(i,0);
        k=0&mr&nk&m[pid]>i:      (m[pid],n[pid]):=(0,0);
        k=0&mr&nk&m[pid]<i&ms:   (m[nid],n[nid],m[pid],n[pid]):=(m[pid],0,0,0);
        k=0&mr&nk&m[pid]=i&ms:   (m[nid],n[nid],m[pid],n[pid],k):=(i,1,0,0,1);
        k=0&mr&!nk&ms:      (m[nid],n[nid],m[pid],n[pid],k):=(m[pid],1,0,0,1);
        k=1&mr:                  (m[pid],n[pid]):=(0,0);
FAIRNESS
        k=1|m[nid]!=0;
```

The fairness property tells that a process cannot stay active (k=0) while letting the message variable of the next process be empty forever. With this additional requirement, the verification results are as follows.

Property	Conclusion
AG(!(p1.k=1)|(AF(p2.n=1)|(p2.k=1)))	true
A(p1.k=0 U AG(p1.k=1))	true

It must be careful of choosing fairness conditions. If it is too strong or inappropriate, then the verification results will be meaningless. For instance, if we put m[(pid+1)%3]!=0, then the verification results will not be useful, since no infinite execution sequences (a finite execution is regarded as infinite with the last state repeated forever, if none of the conditions of the transition rules is satisfiable) satisfy the fairness requirement.

2.4. Specifying Parameterized Systems

To specify a parameterized system where a set of processes are identical, one may use a parameter to represent the number of such processes, as well as parameters to represent the size of array variables and ranges of variables, such that only few changes need to be made for specification of a parameterized systems with different sizes.

The leader election protocol with 3 processes may be specified as follows.

```
VVM     le005
DEFINE
```

```
        N       =3
        NL      =2
        ic      =(p[0].i!=p[1].i)&(p[0].i!=p[2].i)&(p[1].i!=p[2].i)
        nid    =(pid+1)%N
        mr      =(m[pid]!=0)
        ms      =(m[nid]=0)
        nk      =(n[pid]=0)
VAR
        m[0..NL]:0..N;
        n[0..NL]:0..1;
INIT
        ic;
PROC
        p[0..NL]:p0m();
SPEC
        AG(!(p[0].k=1)|(AF(n[1]=1)|(p[1].k=1)));
        A(p[0].k=0 U AG p[0].k=1);

MODULE  p0m()
VAR
        k:0..1;
        i:0..N;
INIT
        k=0;
        m[pid]=0;
        n[pid]=0;
        i>=1;
        i<=N;
TRANS
        k=0&!mr&ms:             (m[nid],n[nid]):=(i,0);
        k=0&mr&nk&m[pid]>i:     (m[pid],n[pid]):=(0,0);
        k=0&mr&nk&m[pid]<i&ms:  (m[nid],n[nid],m[pid],n[pid]):=(m[pid],0,0,0);
        k=0&mr&nk&m[pid]=i&ms:  (m[nid],n[nid],m[pid],n[pid],k):=(i,1,0,0,1);
        k=0&mr&!nk&ms:     (m[nid],n[nid],m[pid],n[pid],k):=(m[pid],1,0,0,1);
        k=1&mr:                 (m[pid],n[pid]):=(0,0);
FAIRNESS
        k=1|m[nid]!=0;
```

For the specification of the leader election protocol with 4 processes, we only need to change the values of N, NL and the definition of ic in the DEFINE section in the VVM declaration as shown below.

```
VVM     le006
```

```
DEFINE
        N       =4
        NL      =3
        ic      =(p[0].i!=p[1].i)&(p[0].i!=p[2].i)&(p[1].i!=p[2].i)&\
                 (p[0].i!=p[3].i)&(p[1].i!=p[3].i)&(p[2].i!=p[3].i)
        nid     =(pid+1)%N
        mr      =(m[pid]!=0)
        ms      =(m[nid]=0)
        nk      =(n[pid]=0)
VAR
        m[0..NL]:0..N;
        n[0..NL]:0..1;
INIT
        ic;
PROC
        p[0..NL]:p0m();
SPEC
        AG(!(p[0].k=1)|(AF(n[1]=1)|(p[1].k=1)));
        A(p[0].k=0 U AG p[0].k=1);

MODULE  p0m()
VAR
        k:0..1;
        i:0..N;
INIT
        k=0;
        m[pid]=0;
        n[pid]=0;
        i>=1;
        i<=N;
TRANS
        k=0&!mr&ms:             (m[nid],n[nid]):=(i,0);
        k=0&mr&nk&m[pid]>i:     (m[pid],n[pid]):=(0,0);
        k=0&mr&nk&m[pid]<i&ms:  (m[nid],n[nid],m[pid],n[pid]):=(m[pid],0,0,0);
        k=0&mr&nk&m[pid]=i&ms:  (m[nid],n[nid],m[pid],n[pid],k):=(i,1,0,0,1);
        k=0&mr&!nk&ms:     (m[nid],n[nid],m[pid],n[pid],k):=(m[pid],1,0,0,1);
        k=1&mr:                 (m[pid],n[pid]):=(0,0);
FAIRNESS
        k=1|m[nid]!=0;
```

2.5. Verification with the －bcc Option

This option is for the use of bounded model checking. It is currently used for checking CTL formulas for models without fairness constraints. To check whether the i-th property holds in the model specified in “le001.vvm” with bounded model checking in which *minisat* is used as the SAT-solver, we may use the command

```
verds -bcc -ck i -satsolver /home/zwh/bin/minisat le001.vvm
```

with a specified value of i. The output of checking the first property is as follows:

```
system_prompt> verds -bcc -ck 1 -satsolver /home/zwh/bin/minisat le001.vvm
VERSION:    verds 1.42 - DEC 2012
FILE:       le001.vvm
PROPERTY:   A G (! (p0.k = 1 )| (A F (p1.n = 1 )| (p1.k = 1 )))
SATSOLVER:  /home/zwh/bin/minisat
INFO:       applying /home/zwh/bin/minisat
bound =  0  time = 1
----------  time = 1
bound =  1  time = 1
----------  time = 1
bound =  2  time = 1
----------  time = 1
bound =  3  time = 1
----------  time = 1
bound =  4  time = 1
----------  time = 1
bound =  5  time = 1
----------  time = 1
bound =  6  time = 1
----------  time = 1
bound =  7  time = 1
----------  time = 2
bound =  8  time = 2
----------  time = 2
bound =  9  time = 2
----------  time = 2
bound = 10  time = 2
----------  time = 2
bound = 11  time = 3
----------  time = 3
bound = 12  time = 3
```

```
----------  time = 3
bound = 13  time = 4
----------  time = 4
bound = 14  time = 4
----------  time = 5
bound = 15  time = 5
----------  time = 6
bound = 16  time = 6
----------  time = 7
bound = 17  time = 7
----------  time = 8
bound = 18  time = 9
----------  time = 10
bound = 19  time = 11
----------  time = 12
bound = 20  time = 14
----------  time = 15
bound = 21  time = 17
CONCLUSION: TRUE (time=17 bound=21)
```

It is similar with checking the second property, and the result is as follows:

```
system_prompt>  verds -bcc -ck 2 -satsolver /home/zwh/bin/minisat le001.vvm
VERSION:    verds 1.42 - DEC 2012
FILE:       le001.vvm
PROPERTY:   A ((p0.k = 0 )U A G (p0.k = 1 ))
SATSOLVER:  /home/zwh/bin/minisat
INFO:       applying /home/zwh/bin/minisat
bound =  0  time = 0
----------  time = 0
bound =  1  time = 1
----------  time = 1
bound =  2  time = 1
----------  time = 1
CONCLUSION: FALSE (time=1 bound=2)
```