Exhaustive Analysis of a Mutual Exclusion Algorithm

Gerard J. Holzmann
MH 11271 Phone 6335 Room 2C-521
research!gerard
9 June 1987

Abstract

With minor modifications a reachability analysis technique that was developed for tracing design errors in formalized descriptions of communication protocols, can be used to verify properties of mutual exclusion algorithms. The paper gives an example of a mutual exclusion algorithm, proven correct by its author with an informal argument, and shown to be erroneous with relatively little effort by an exhaustive symbolic execution. For a three process model, the longest execution run performed, took 13 seconds of CPU time on a VAX-11/750, and produced 13 error sequences. The 13 sequences revealed three different types of error in the algorithm.

1. Introduction

A mutual exclusion algorithm is meant to provide concurrent processes a mutually exclusive access to a common critical region in their code, while relying only on the atomicity of individual read and write operations. A first solution to this problem, for 2 processes, is attributed to the Dutch mathematician Dekker (1962). The solution was generalized for *N* processes by Dijkstra, and published in 1968 [Dijkstra '68]. Every few years since then, an improvement or an alternative solution is published, accompanied by a correctness argument that is not always more understandable than the algorithm it describes. For a recent overview see e.g. [Lamport '86].

One such improved algorithm was submitted to a journal and then to me for review. Because a validation of the algorithm seemed easier than a validation of its proof, I constructed the following model in the validation language *Argos* [Holz '87a].

```
#define N    3   /* the number of concurrent processes */

pvar time, someone_in, critical;
pvar  req[N];
pvar cand[N];

mutex(id)
{   pvar n, m;

    req[id] = time;
    n = (id+1)%N;
    m = 0;
    do
    :: (m <  N-1) -> (req[n] >= req[id]) -> m++; n = (n+1)%N
    :: (m == N-1) -> break
    od;
L2: (someone_in == 0);
    cand[id] = 1;
    if
    :: (someone_in == 1) -> cand[id] = 0; goto L2
    :: (someone_in == 0) -> someone_in = 1
    fi;
    n = id+1;
    do
    :: (n <  N) -> cand[n] = 0; n++
    :: (n >= N) -> break
    od;

L3: if
    :: (cand[id] == 0) -> goto L2
    :: (cand[id] == 1) -> skip
    fi;
    n = 0;

    do
    :: (n <  id) ->
        if
        :: (cand[n] == 1) -> goto L3
        :: (cand[n] == 0) -> n++
        fi
    :: (n >= id) -> break
    od;


    critical++; (critical == 1); critical--;

    cand[id] = 0;
    req[id] = 100;
    time++;
    someone_in = 0
}


proc P[N]    /* an array of N processes */
{
    mutex(_PROCID)
}
```

2. Intended working

The *Argos* program above contains a procedure declaration *mutex*(*id*) and a declaration for an array of *N* processes *P*[*N*]. *N* is a constant defined in the first line of the program. There are three global variables, named *time*, *someone_in*, and *critical*, and two global arrays of *N* variables each, named *req*[*N*] and *cand*[*N*]. Each process has a unique identity between 0 and $N - 1$. The identity is available as a predefined variable *_PROCID*, which is passed by each process to procedure *mutex*.

In the first line of *mutex*, the process makes a request for access to the critical region by setting its slot in array *req* to the value of a variable *time*. This variable is incremented each time a process leaves its critical region. The next six lines in *mutex* are meant give priority to competing processes that selected a lower value of *time*, i.e. to processes that arrived earlier than the current process. The *do* loop lists two options (preceded by double colons). An option within such a loop can only be selected for execution if its first statement (immediately following the double colon) is ‘executable,’ or in this case: if the condition evaluates to *true*. The loop terminates when the *break* statement is executed.

At label $L2$ a single condition is listed. In *Argos* statements a small set of rules determines if statements are executable or not. Assignments, for instance, are always executable, but conditions are only executable if they are true. If a non-executable statement is the only option that a process has to continue, that process will block. In this case, the executing process will block until ($someone\_in \equiv 0$). It will then announce itself as a candidate for entering the critical region, and will have to compete for access with those processes (if any) that selected the same value of *time*.

A second check for the value of *someone_in* is performed, and if the value has changed, the announcement via $cand[id]$ is undone and the wait at label $L2$ is resumed. If the second test is passed successfully, *someone_in* is now set to 1, thus blocking further attempts to pass to this part of the algorithm, and a check is done on the presence of competing processes with a *higher* id number that succeeded in announcing their candidacy for entry through *cand*. If any of these processes is found, their entry in *cand* is reset. Another test follows to see if no other process has reset the current process’s candidacy. Then a final check is done on competing processes with a *lower* id number. These processes take precedence in entering their critical section, and should succeed in turning the current process’ entry in *cand* off.

Once the critical section is entered, a count is incremented. The count should never have any other value than 1 once the process has obtained exclusive access to this portion of the code. A test ($critical \equiv 1$) will block, and cause a deadlock, if this assertion is violated.

Upon exit from the critical section, the variables used are reset to their initial values, and for the purposes of this test, the entry in *req* is set to a non-competing value.

3. Analysis

The algorithm is complex enough to make it not entirely obvious that it either will or will not work as advertised. The version above can be compiled into an extended FSM model, and analyzed for the absence of deadlocks by a conventional protocol analyzer such as ‘trace’ [Holz ’87b] if some of its search heuristics are turned off. Trace is optimized for synchronization via message passing. Access to a variable can then always be interpreted as an internal step, that cannot affect the progress of competing processes. Here, variables are global, and deliberately used to enforce the synchronization. A purely exhaustive search mode was added to ‘trace’ to allow for an analysis of this specific problem. For $N = 2$, trace finds a violation of the mutual exclusion requirement in 12 seconds of CPU time on a VAX-8550, or in 134 seconds on a VAX-11/750 (see also below). Though this first test already proves the inadequacy of the algorithm, it is interesting to test the behavior also for larger numbers of processes. For $N > 2$, the number of cases to analyze in an exhaustive symbolic execution, however, grows very rapidly.

The algorithm can also be rewritten without procedure calls and arrays, so that it can be translated into a state vector model and validated more efficiently with *supertrace* [Holz ’87c]. For $N = 2$ the rewritten algorithm is analyzed exhaustively with *supertrace* in 0.1 seconds of CPU time on a VAX-8550, or in 1.4 seconds on a VAX-11/750, producing the same deadlock revealed by *trace* before. For $N = 3$ the algorithm becomes:

```
pvar time, someone_in, critical;
pvar req_0, req_1, req_2;
pvar cand_0, cand_1, cand_2;

proc P0
{
    req_0 = time;
    (req_0 <= req_1 && req_0 <= req_2);
```

```
L2: (someone_in == 0);
    cand_0 = 1;
    if
    :: (someone_in == 1) -> cand_0 = 0; goto L2
    :: (someone_in == 0) -> someone_in = 1
    fi;
    cand_1 = 0;  /* cand_n n > i */
    cand_2 = 0;
    if
    :: (cand_0 == 0) -> goto L2
    :: (cand_0 == 1) -> skip
    fi;

            /* if cand_n -> goto L3 n < i */
    critical++;
    (critical == 1);
    critical--;


    cand_0 = 0;
    req_0 = 100;
    time++;
    someone_in = 0;
end0:   skip
}


proc P1
{
    req_1 = time;
    (req_1 <= req_0 && req_1 <= req_2);
L2: (someone_in == 0);
    cand_1 = 1;
    if
    :: (someone_in == 1) -> cand_1 = 0; goto L2
    :: (someone_in == 0) -> someone_in = 1
    fi;

    cand_2 = 0;
L3: if
    :: (cand_1 == 0) -> goto L2
    :: (cand_1 == 1) -> skip
    fi;
    if
    :: (cand_0 == 1) -> goto L3      /* n < i */
    :: (cand_0 == 0) -> skip
    fi;


    critical++;
    (critical == 1);
    critical--;
    cand_1 = 0;
    req_1 = 100;
    time++;
    someone_in = 0;
end1:   skip
}


proc P2
{
    req_2 = time;
    (req_2 <= req_0 && req_2 <= req_1);
L2: (someone_in == 0);
    cand_2 = 1;
    if
    :: (someone_in == 1) -> cand_2 = 0; goto L2
    :: (someone_in == 0) -> someone_in = 1
    fi;


L3: if
    :: (cand_2 == 0) -> goto L2
    :: (cand_2 == 1) -> skip
    fi;
    if
    :: (cand_0 == 1 || cand_1 == 1) -> goto L3        /* n < i */
    :: (cand_0 == 0 && cand_1 == 0) -> skip
    fi;


    critical++;
    (critical == 1);
    critical--;


    cand_2 = 0;
    req_2 = 100;
    time++;
    someone_in = 0;
end2:   skip
}
```

This 3-process specification is analyzed exhaustively by *supertrace* in 1.2 seconds on a VAX-8550, (12.9 seconds on the VAX-11/750), generating 6,184 unique states of which 13 are flagged as deadlocks. The longest unique execution path is 55 steps long (one step is a single assignment or boolean condition).

4. Types of Error

In four cases, the mutual exclusion requirement is violated, and the lock occurs on condition (*critical* ≡ 1). In four other cases, two processes succeed in reaching their endstates, while the third process gets stuck in an infinite wait at label *L*2. In the remaining five error sequences just one process reaches its end state, leaving the two other process blocked. The complete error sequences, as reported by *supertrace* are listed in the appendix. Each error is preceded by a line that prints the value of all variables at the time of the lock. The variables are given in the following order:

```
G{time,someone_in,req_0,req_1,req_2,cand_0,cand_1,cand_2,critical,}
```

The error itself is given as a sequence of events. The first number followed by a colon is the step number. The second number is the identity of the process executing the step (0, 1 or 2). The remainder of the line is the source text of the event executed. Braces have no particular significance.

5. A Working Algorithm

Out of curiosity, the following algorithm, presented in [Lamport '86], was also validated with an exhaustive search. The algorithm is called "the one-bit algorithm". In *Argos* the specification looks as follows.

```
#define N    3

#define false    0
#define true 1

pvar x[N];
pvar critical = 0;

mutex(id)
{   pvar j;

again:  x[id] = true;
    j = 0;
    do
    :: (j < id) ->   if
            :: (x[j] == true) -> x[id] = false;
                    (x[j] == false);
                    goto again
            :: (x[j] == false) -> skip
            fi;
            j++
    :: (j >= id) -> break
    od;
    j = id+1;
    do
    :: (j <  N) -> (x[j] == false)
    :: (j >= N) -> break
    od;
    critical++;
    (critical == 1);
    critical--;
    x[id] = false;
    goto again
}

proc compete[N]
{
    mutex(_PROCID)
}
```

For two processes the exhaustive validation (with the conventional validator) takes 6.12 seconds on a VAX-11/750. For three processes the validation takes approximately 17 minutes. As expected, for this algorithm no errors are reported.

References

[Dijkstra ’65] Dijkstra, E.W. (1965), "Solution of a problem in concurrent programming control," CACM, Vol. 8, No. 9, Sept. 1965, p. 569.

[Holz ’87a] Holzmann, G.J., "Manual for the protocol analyzer ‘trace,’ AT&T Bell Laboratories, Computing Science Technical Report No. 134, February 1987, 27 pgs.

[Holz ’87b] Holzmann, G.J., "Automated Protocol Validation in Argos: Assertion Proving ans Scatter Searching," IEEE Trans. on Software Engineering, Vol. 13, No. 6, June 1987.

[Holz ’87c] Holzmann, G.J., "An improved protocol reachability analysis technique," AT&T Bell Labs, TM 11271-870527-07, May 27, 1987, 18 pgs.

[Lamport ’86] Lamport, L. (1986), "The Mutual Exclusion Problem − parts I and II", Journal of the ACM, Vol. 33, No. 2, April 1986, pp. 313-347.

APPENDIX

The 13 Error Sequences

```
G{2,1,4,4,1,0,0,0,0,}    deadlock
     0: 0: (req_0 = time)
     1: 0: ((req_0 <= req_1) && (req_0 <= req_2))
     2: 0: (someone_in == 0)
     3: 0: (cand_0 = 1)
     4: 0: (someone_in == 0)
     5: 0: (someone_in = 1)
     6: 0: (cand_1 = 0)
     7: 0: (cand_2 = 0)
     8: 0: (cand_0 == 1)
     9: 0: critical = critical + 1
    10: 0: (critical == 1)
    11: 0: critical = critical - 1
    12: 0: (cand_0 = 0)
    13: 0: (req_0 = 100)
    14: 0: time = time + 1
    15: 0: (someone_in = 0)
    16: 1: (req_1 = time)
    17: 2: (req_2 = time)
    18: 1: ((req_1 <= req_0) && (req_1 <= req_2))
    19: 1: (someone_in == 0)
    20: 1: (cand_1 = 1)
    21: 1: (someone_in == 0)
    22: 2: ((req_2 <= req_0) && (req_2 <= req_1))
    23: 2: (someone_in == 0)
    24: 1: (someone_in = 1)
    25: 2: (cand_2 = 1)
    26: 1: (cand_2 = 0)
    27: 1: (cand_1 == 1)
    28: 1: (cand_0 == 0)
    29: 1: critical = critical + 1
    30: 1: (critical == 1)
    31: 1: critical = critical - 1
    32: 1: (cand_1 = 0)
    33: 1: (req_1 = 100)
    34: 1: time = time + 1
    35: 1: (someone_in = 0)
    36: 2: (someone_in == 0)
    37: 2: (someone_in = 1)
    38: 2: (cand_2 == 0)
```

```
G{2,1,4,4,0,0,0,0,0,}    deadlock
     0: 0: (req_0 = time)
     1: 0: ((req_0 <= req_1) && (req_0 <= req_2))
     2: 0: (someone_in == 0)
     3: 0: (cand_0 = 1)
     4: 0: (someone_in == 0)
     5: 0: (someone_in = 1)
     6: 0: (cand_1 = 0)
     7: 0: (cand_2 = 0)
     8: 0: (cand_0 == 1)
     9: 0: critical = critical + 1
    10: 0: (critical == 1)
    11: 0: critical = critical - 1
    12: 0: (cand_0 = 0)
    13: 0: (req_0 = 100)
    14: 1: (req_1 = time)
    15: 1: ((req_1 <= req_0) && (req_1 <= req_2))
    16: 2: (req_2 = time)
    17: 0: time = time + 1
    18: 0: (someone_in = 0)
    19: 1: (someone_in == 0)
    20: 1: (cand_1 = 1)
    21: 1: (someone_in == 0)
    22: 2: ((req_2 <= req_0) && (req_2 <= req_1))
    23: 2: (someone_in == 0)
    24: 1: (someone_in = 1)
    25: 2: (cand_2 = 1)
    26: 1: (cand_2 = 0)
    27: 1: (cand_1 == 1)
    28: 1: (cand_0 == 0)
    29: 1: critical = critical + 1
    30: 1: (critical == 1)
    31: 1: critical = critical - 1
    32: 1: (cand_1 = 0)
    33: 1: (req_1 = 100)
    34: 1: time = time + 1
    35: 1: (someone_in = 0)
    36: 2: (someone_in == 0)
    37: 2: (someone_in = 1)
    38: 2: (cand_2 == 0)
```

```
G{1,1,4,0,1,0,0,0,0,}    deadlock
     0: 0: (req_0 = time)
     1: 0: ((req_0 <= req_1) && (req_0 <= req_2))
     2: 0: (someone_in == 0)
     3: 0: (cand_0 = 1)
     4: 0: (someone_in == 0)
     5: 1: (req_1 = time)
     6: 1: ((req_1 <= req_0) && (req_1 <= req_2))
     7: 1: (someone_in == 0)
     8: 0: (someone_in = 1)
     9: 1: (cand_1 = 1)
    10: 0: (cand_1 = 0)
    11: 0: (cand_2 = 0)
    12: 0: (cand_0 == 1)
    13: 0: critical = critical + 1
    14: 0: (critical == 1)
    15: 0: critical = critical - 1
    16: 0: (cand_0 = 0)
    17: 0: (req_0 = 100)
    18: 0: time = time + 1
    19: 0: (someone_in = 0)
    20: 1: (someone_in == 0)
    21: 1: (someone_in = 1)
    22: 1: (cand_2 = 0)
    23: 1: (cand_1 == 0)
    24: 2: (req_2 = time)
```

```
G{1,1,4,0,0,0,0,0,0,}    deadlock
     0: 0: (req_0 = time)
     1: 0: ((req_0 <= req_1) && (req_0 <= req_2))
     2: 0: (someone_in == 0)
     3: 0: (cand_0 = 1)
     4: 0: (someone_in == 0)
     5: 1: (req_1 = time)
     6: 1: ((req_1 <= req_0) && (req_1 <= req_2))
     7: 1: (someone_in == 0)
     8: 0: (someone_in = 1)
     9: 1: (cand_1 = 1)
    10: 0: (cand_1 = 0)
    11: 0: (cand_2 = 0)
    12: 0: (cand_0 == 1)
    13: 0: critical = critical + 1
    14: 0: (critical == 1)
    15: 0: critical = critical - 1
    16: 0: (cand_0 = 0)
    17: 0: (req_0 = 100)
    18: 2: (req_2 = time)
    19: 0: time = time + 1
    20: 0: (someone_in = 0)
    21: 1: (someone_in == 0)
    22: 1: (someone_in = 1)
    23: 1: (cand_2 = 0)
    24: 1: (cand_1 == 0)
    25: 2: ((req_2 <= req_0) && (req_2 <= req_1))
```

```
G{2,1,4,0,4,0,0,0,0,}    deadlock
     0: 0: (req_0 = time)
     1: 0: ((req_0 <= req_1) && (req_0 <= req_2))
     2: 0: (someone_in == 0)
     3: 0: (cand_0 = 1)
     4: 0: (someone_in == 0)
     5: 1: (req_1 = time)
     6: 1: ((req_1 <= req_0) && (req_1 <= req_2))
     7: 1: (someone_in == 0)
     8: 0: (someone_in = 1)
     9: 1: (cand_1 = 1)
    10: 0: (cand_1 = 0)
    11: 0: (cand_2 = 0)
    12: 0: (cand_0 == 1)
    13: 0: critical = critical + 1
    14: 0: (critical == 1)
    15: 0: critical = critical - 1
    16: 0: (cand_0 = 0)
    17: 0: (req_0 = 100)
    18: 2: (req_2 = time)
    19: 0: time = time + 1
    20: 0: (someone_in = 0)
    21: 1: (someone_in == 0)
    22: 2: ((req_2 <= req_0) && (req_2 <= req_1))
    23: 2: (someone_in == 0)
    24: 2: (cand_2 = 1)
    25: 2: (someone_in == 0)
    26: 2: (someone_in = 1)
    27: 2: (cand_2 == 1)
    28: 2: ((cand_0 == 0) && (cand_1 == 0))
    29: 2: critical = critical + 1
    30: 2: (critical == 1)
    31: 2: critical = critical - 1
    32: 2: (cand_2 = 0)
    33: 2: (req_2 = 100)
    34: 2: time = time + 1
    35: 2: (someone_in = 0)
    36: 1: (someone_in = 1)
    37: 1: (cand_2 = 0)
    38: 1: (cand_1 == 0)
```

```
G{1,1,4,0,0,0,0,1,2,}    deadlock
     0: 0: (req_0 = time)
     1: 0: ((req_0 <= req_1) && (req_0 <= req_2))
     2: 0: (someone_in == 0)
     3: 0: (cand_0 = 1)
     4: 0: (someone_in == 0)
     5: 1: (req_1 = time)
     6: 1: ((req_1 <= req_0) && (req_1 <= req_2))
     7: 1: (someone_in == 0)
     8: 1: (cand_1 = 1)
     9: 1: (someone_in == 0)
    10: 0: (someone_in = 1)
    11: 1: (someone_in = 1)
    12: 1: (cand_2 = 0)
    13: 1: (cand_1 == 1)
    14: 0: (cand_1 = 0)
    15: 0: (cand_2 = 0)
    16: 0: (cand_0 == 1)
    17: 0: critical = critical + 1
    18: 0: (critical == 1)
    19: 0: critical = critical - 1
    20: 0: (cand_0 = 0)
    21: 0: (req_0 = 100)
    22: 1: (cand_0 == 0)
    23: 1: critical = critical + 1
    24: 2: (req_2 = time)
    25: 0: time = time + 1
    26: 0: (someone_in = 0)
    27: 2: ((req_2 <= req_0) && (req_2 <= req_1))
    28: 2: (someone_in == 0)
    29: 2: (cand_2 = 1)
    30: 2: (someone_in == 0)
    31: 2: (someone_in = 1)
    32: 2: (cand_2 == 1)
    33: 2: ((cand_0 == 0) && (cand_1 == 0))
    34: 2: critical = critical + 1
```

```
G{1,1,4,0,0,0,1,0,2,}    deadlock
     0: 0: (req_0 = time)
     1: 0: ((req_0 <= req_1) && (req_0 <= req_2))
     2: 0: (someone_in == 0)
     3: 0: (cand_0 = 1)
     4: 0: (someone_in == 0)
     5: 1: (req_1 = time)
     6: 1: ((req_1 <= req_0) && (req_1 <= req_2))
     7: 1: (someone_in == 0)
     8: 1: (cand_1 = 1)
     9: 1: (someone_in == 0)
    10: 2: (req_2 = time)
    11: 2: ((req_2 <= req_0) && (req_2 <= req_1))
    12: 2: (someone_in == 0)
    13: 2: (cand_2 = 1)
    14: 2: (someone_in == 0)
    15: 0: (someone_in = 1)
    16: 0: (cand_1 = 0)
    17: 1: (someone_in = 1)
    18: 2: (someone_in = 1)
    19: 2: (cand_2 == 1)
    20: 0: (cand_2 = 0)
    21: 0: (cand_0 == 1)
    22: 0: critical = critical + 1
    23: 0: (critical == 1)
    24: 0: critical = critical - 1
    25: 0: (cand_0 = 0)
    26: 0: (req_0 = 100)
    27: 0: time = time + 1
    28: 0: (someone_in = 0)
    29: 1: (cand_2 = 0)
    30: 1: (cand_1 == 0)
    31: 1: (someone_in == 0)
    32: 2: ((cand_0 == 0) && (cand_1 == 0))
    33: 1: (cand_1 = 1)
    34: 1: (someone_in == 0)
    35: 1: (someone_in = 1)
    36: 1: (cand_2 = 0)
    37: 1: (cand_1 == 1)
    38: 1: (cand_0 == 0)
    39: 1: critical = critical + 1
    40: 2: critical = critical + 1
```

```
G{1,0,4,0,0,0,0,0,2,}    deadlock
     0: 0: (req_0 = time)
     1: 0: ((req_0 <= req_1) && (req_0 <= req_2))
     2: 0: (someone_in == 0)
     3: 0: (cand_0 = 1)
     4: 0: (someone_in == 0)
     5: 1: (req_1 = time)
     6: 1: ((req_1 <= req_0) && (req_1 <= req_2))
     7: 1: (someone_in == 0)
     8: 1: (cand_1 = 1)
     9: 1: (someone_in == 0)
    10: 2: (req_2 = time)
    11: 2: ((req_2 <= req_0) && (req_2 <= req_1))
    12: 2: (someone_in == 0)
    13: 2: (cand_2 = 1)
    14: 2: (someone_in == 0)
    15: 0: (someone_in = 1)
    16: 1: (someone_in = 1)
    17: 2: (someone_in = 1)
    18: 2: (cand_2 == 1)
    19: 1: (cand_2 = 0)
    20: 1: (cand_1 == 1)
    21: 0: (cand_1 = 0)
    22: 0: (cand_2 = 0)
    23: 0: (cand_0 == 1)
    24: 0: critical = critical + 1
    25: 0: (critical == 1)
    26: 0: critical = critical - 1
    27: 0: (cand_0 = 0)
    28: 0: (req_0 = 100)
    29: 0: time = time + 1
    30: 0: (someone_in = 0)
    31: 1: (cand_0 == 0)
    32: 1: critical = critical + 1
    33: 2: ((cand_0 == 0) && (cand_1 == 0))
    34: 2: critical = critical + 1
```

```
G{1,1,4,1,0,0,0,0,0,}    deadlock
     0: 0: (req_0 = time)
     1: 0: ((req_0 <= req_1) && (req_0 <= req_2))
     2: 0: (someone_in == 0)
     3: 0: (cand_0 = 1)
     4: 0: (someone_in == 0)
     5: 2: (req_2 = time)
     6: 2: ((req_2 <= req_0) && (req_2 <= req_1))
     7: 2: (someone_in == 0)
     8: 0: (someone_in = 1)
     9: 0: (cand_1 = 0)
    10: 2: (cand_2 = 1)
    11: 0: (cand_2 = 0)
    12: 0: (cand_0 == 1)
    13: 0: critical = critical + 1
    14: 0: (critical == 1)
    15: 0: critical = critical - 1
    16: 0: (cand_0 = 0)
    17: 0: (req_0 = 100)
    18: 0: time = time + 1
    19: 0: (someone_in = 0)
    20: 1: (req_1 = time)
    21: 2: (someone_in == 0)
    22: 2: (someone_in = 1)
    23: 2: (cand_2 == 0)
```

```
G{1,1,0,4,0,0,0,0,0,}    deadlock
     0: 0: (req_0 = time)
     1: 0: ((req_0 <= req_1) && (req_0 <= req_2))
     2: 0: (someone_in == 0)
     3: 0: (cand_0 = 1)
     4: 1: (req_1 = time)
     5: 1: ((req_1 <= req_0) && (req_1 <= req_2))
     6: 1: (someone_in == 0)
     7: 1: (cand_1 = 1)
     8: 1: (someone_in == 0)
     9: 2: (req_2 = time)
    10: 2: ((req_2 <= req_0) && (req_2 <= req_1))
    11: 2: (someone_in == 0)
    12: 1: (someone_in = 1)
    13: 0: (someone_in == 1)
    14: 0: (cand_0 = 0)
    15: 2: (cand_2 = 1)
    16: 1: (cand_2 = 0)
    17: 1: (cand_1 == 1)
    18: 1: (cand_0 == 0)
    19: 1: critical = critical + 1
    20: 1: (critical == 1)
    21: 1: critical = critical - 1
    22: 1: (cand_1 = 0)
    23: 1: (req_1 = 100)
    24: 1: time = time + 1
    25: 1: (someone_in = 0)
    26: 0: (someone_in == 0)
    27: 0: (cand_0 = 1)
    28: 2: (someone_in == 0)
    29: 2: (someone_in = 1)
    30: 0: (someone_in == 1)
    31: 0: (cand_0 = 0)
    32: 2: (cand_2 == 0)
```

```
G{1,1,0,4,0,1,0,0,2,}    deadlock
      0: 0: (req_0 = time)
      1: 0: ((req_0 <= req_1) && (req_0 <= req_2))
      2: 0: (someone_in == 0)
      3: 0: (cand_0 = 1)
      4: 1: (req_1 = time)
      5: 1: ((req_1 <= req_0) && (req_1 <= req_2))
      6: 1: (someone_in == 0)
      7: 1: (cand_1 = 1)
      8: 1: (someone_in == 0)
      9: 2: (req_2 = time)
     10: 2: ((req_2 <= req_0) && (req_2 <= req_1))
     11: 2: (someone_in == 0)
     12: 2: (cand_2 = 1)
     13: 2: (someone_in == 0)
     14: 1: (someone_in = 1)
     15: 0: (someone_in == 1)
     16: 0: (cand_0 = 0)
     17: 2: (someone_in = 1)
     18: 2: (cand_2 == 1)
     19: 1: (cand_2 = 0)
     20: 1: (cand_1 == 1)
     21: 1: (cand_0 == 0)
     22: 1: critical = critical + 1
     23: 1: (critical == 1)
     24: 1: critical = critical - 1
     25: 1: (cand_1 = 0)
     26: 1: (req_1 = 100)
     27: 1: time = time + 1
     28: 1: (someone_in = 0)
     29: 0: (someone_in == 0)
     30: 2: ((cand_0 == 0) && (cand_1 == 0))
     31: 0: (cand_0 = 1)
     32: 0: (someone_in == 0)
     33: 0: (someone_in = 1)
     34: 0: (cand_1 = 0)
     35: 0: (cand_2 = 0)
     36: 0: (cand_0 == 1)
     37: 0: critical = critical + 1
     38: 2: critical = critical + 1
```

```
G{1,1,1,4,0,0,0,0,0,}    deadlock
     0: 1: (req_1 = time)
     1: 1: ((req_1 <= req_0) && (req_1 <= req_2))
     2: 1: (someone_in == 0)
     3: 1: (cand_1 = 1)
     4: 1: (someone_in == 0)
     5: 2: (req_2 = time)
     6: 2: ((req_2 <= req_0) && (req_2 <= req_1))
     7: 2: (someone_in == 0)
     8: 1: (someone_in = 1)
     9: 2: (cand_2 = 1)
    10: 1: (cand_2 = 0)
    11: 1: (cand_1 == 1)
    12: 1: (cand_0 == 0)
    13: 1: critical = critical + 1
    14: 1: (critical == 1)
    15: 1: critical = critical - 1
    16: 1: (cand_1 = 0)
    17: 1: (req_1 = 100)
    18: 1: time = time + 1
    19: 0: (req_0 = time)
    20: 1: (someone_in = 0)
    21: 2: (someone_in == 0)
    22: 2: (someone_in = 1)
    23: 2: (cand_2 == 0)
```

```
G{2,1,4,1,4,0,0,0,0,}    deadlock
     0: 2: (req_2 = time)
     1: 2: ((req_2 <= req_0) && (req_2 <= req_1))
     2: 2: (someone_in == 0)
     3: 2: (cand_2 = 1)
     4: 2: (someone_in == 0)
     5: 2: (someone_in = 1)
     6: 2: (cand_2 == 1)
     7: 2: ((cand_0 == 0) && (cand_1 == 0))
     8: 2: critical = critical + 1
     9: 2: (critical == 1)
    10: 2: critical = critical - 1
    11: 2: (cand_2 = 0)
    12: 2: (req_2 = 100)
    13: 2: time = time + 1
    14: 0: (req_0 = time)
    15: 1: (req_1 = time)
    16: 0: ((req_0 <= req_1) && (req_0 <= req_2))
    17: 1: ((req_1 <= req_0) && (req_1 <= req_2))
    18: 2: (someone_in = 0)
    19: 0: (someone_in == 0)
    20: 0: (cand_0 = 1)
    21: 0: (someone_in == 0)
    22: 1: (someone_in == 0)
    23: 0: (someone_in = 1)
    24: 1: (cand_1 = 1)
    25: 0: (cand_1 = 0)
    26: 0: (cand_2 = 0)
    27: 0: (cand_0 == 1)
    28: 0: critical = critical + 1
    29: 0: (critical == 1)
    30: 0: critical = critical - 1
    31: 0: (cand_0 = 0)
    32: 0: (req_0 = 100)
    33: 0: time = time + 1
    34: 0: (someone_in = 0)
    35: 1: (someone_in == 0)
    36: 1: (someone_in = 1)
    37: 1: (cand_2 = 0)
    38: 1: (cand_1 == 0)
```