Project 2: SPIN model checking of a social network

Beerend Lauwers and Frank Wijmans

April 13, 2012

1 Introduction

The minimal problem to solve was as follows: Build a model for a social network for 4 users with the following social relation:

- User 0 has no followers
- User n has two followers, i+x mod 4, for x = 1 and x = 2.

Some restrictions were applied to minimize the state space. Furthermore, the requirements for correctness were:

1. Privacy constraints
2. No deadlocking
3. Messages will eventually reach its recipients.

2 Data definitions

In the model we use the following global definitions and data types to build our algorithm upon. We define the number of users as 4 and of those four users, one is non-deterministically chosen to send a message.

As for data types, we use channel arrays containing messages of the form of $byte, byte, byte$. The use of channels was required. We have two channels per user, namely an inbox and an outbox. The relation between a user and its channels are identified by the index of the array.

User 1 has $inbox[1]$ and $outbox[1]$ as channels.

3 Algorithm

Our algorithm is based on the following idea: *All users have one device, it is a* `proctype` *called userDevice which takes a user identification byte as its single parameter.* Each user device does three things which are looped in non-deterministic fashion using the `do` notation.

- **Creating new messages:** If both the user's inbox and outbox is empty, a message is added to the outbox of the same user.

- **Reading inbox messages:** If the inbox is not empty, a message is removed from the inbox, and if the constraint requires the user to forward the message, then try and forward.

    1. **Forwarding messages:** If it is possible to add another message to the outbox, do so. Otherwise, place the message back in the inbox and try again later. (It will wait for the outbox to be emptied.)

- **Sending messages:** If the outbox is not empty, take an outbox message, and add the message to the inbox of the direct followers of the user.

4 Linear time logic formulae

We have two LTL formulae to verify the most difficult correctness requirement. Namely the requirement that messages eventually reach the followers in the case of an *Fo* message, and reach the followers, and followers of followers if it is a *Fo2* message.

Formulating these requirements in one LTL formula would be an large statement, and would only make it less readable.

We have chosen to define two formulae because we argue that when two runs, one for each of the formulae shows that respective formula is verified, it would imply that both hold as if it were one statement.

First, we defined the formula for *Fo*. The LTL-formula below is to check whether a sent message (at label `S0`) is eventually read (at label `R0`) for the first of the two followers by stating $(i+1)\%users$. A similar formula was constructed for the second follower, which uses the label `S1` and refers to $(i+2)\%users$. We quantify over the content in the variable k, it is use to check if the content of two messages is the same. Because we are dealing with a historical value when eventually the message is read, we need to use this helper-variable k.

$$
\begin{aligned}
\forall i, k.\Box\ (i > 0 \\
&\wedge\ device[i]@S0 \\
&\wedge\ device[i] : constraint > 0) \\
&\wedge\ device[i] : content == k) \\
&\rightarrow \\
\Diamond\ (&device[(i+1)\%users]@R0 \\
&\wedge\ device[(i+1)\%users] : content == k)
\end{aligned}
$$

For the constraint of *Fo2* we have a formula that resembles the formula above. It may be clear that we have to not only check the two followers, but also the followers of those two. And we have to make sure that the followers are not user 0, as since user 0 does not have followers, they will never forward a message. Again, the formula below is only a part of the entire formula, where

we check the two followers of a follower of the sender.

$$
\begin{aligned}
\forall i, k. \Box\ & (i > 0 \\
& \quad \wedge\ device[i]@S0 \\
& \quad \wedge\ device[i] : constraint == 2 \\
& \quad \wedge\ device[i] : content == k \\
& \quad \wedge\ ((i+1)\%users) > 0) \\
& \rightarrow \\
& \Diamond\ (device[(((i+1)\%users)+1)\%users]@R0 \\
& \quad \wedge\ device[(((i+1)\%users)+1)\%users] : content == k \\
& \quad device[(((i+1)\%users)+2)\%users]@R0 \\
& \quad \wedge\ device[(((i+1)\%users)+2)\%users] : content == k)
\end{aligned}
$$

5 Problems

During this project we have encountered many problems, some of which made it necessary to start with a clean slate. Some were constructs that weren't allowed by SPIN.

For us, the greatest learning points were in solving the following problems.

5.1 Search depth too small

First we had the problem that we had a enormous search depth. We solved this by limiting the messages we created. We originally created infinitely new messages, which would clog up the system with an infinite amount of interleaving possibilities. We now argue that if it works for some messages, it also will work for many messages, since the algorithm stays the same, even if you scale it up.

5.2 Universal relation or forwarding

Another problem we encountered was the problem of forwarding. Until we presented our first approach we used to send messages directly to the followers of followers, as we assumed that the relation of followers was universally known. After it was specified that we needed to forward messages we found that we needed an in and outbox per user, whereas we used to only have a single inbox per user. This required us to rethink our algorithm and specify how the devices would need to go about using the inbox and outbox without it deadlocking.

5.3 Simulating, but not verifying

At some point we had an algorithm where it would run through a simulation. We could see that the algorithm allowed to send, forward and read messages. The problem was that it would not start a verify run. After backtracking our progress, it was clear that we were not allowed to use an atomic{} block, with

a do loop that would possibly not terminate in it:

$$
\begin{array}{l}
atomic\{ \\
\quad do \\
\quad :: \quad (vi < 5) -> vi++; \\
\quad :: \quad (vi > 1) -> vi++; \\
\quad :: \quad break; \\
\quad od \\
\}
\end{array}
$$

We fixed it by removing the second line so it would always break after a non-deterministic number of increments. A fairly simple solution to a very odd (and time-consuming) problem.

5.4 Linear time logic

We designed LTL formulae, but couldn't check them until verification was possible. After the other problems were solved, we started working on the LTL formulae.

After trying our first formulae we found that we needed an extra mechanism for knowing that all devices had done all the work. We added three statement in the devices' do loop, in which we said:

$$
\begin{array}{l}
::uniqueContent >= MSGLIMIT \wedge \\
\quad empty(inbox[user]) \wedge \\
\quad empty(outbox[user]) \wedge \\
\quad isActive[user] == 1 \\
\rightarrow isActive[user] = 0 \\
\\
::uniqueContent >= MSGLIMIT \wedge \\
\quad isActive[user] == 0 \wedge \\
\quad (isActive[0] == 1 \vee \\
\quad isActive[1] == 1 \vee \\
\quad isActive[2] == 1 \vee \\
\quad isActive[3] == 1) \\
\rightarrow skip; \\
::uniqueContent >= MSGLIMIT \wedge \\
\quad (isActive[0] == 0 \wedge \\
\quad isActive[1] == 0 \wedge \\
\quad isActive[2] == 0 \wedge \\
\quad isActive[3] == 0) \\
\rightarrow break;
\end{array}
$$

The first describes that we set our device to inactive when we have completely empty channels. Secondly, we skip when we are inactive, but someone else is still active. Third, we terminate when all users are inactive.

But this was not enough because the problem would be that when a device clears its last message, and all other devices are idle, it might still be the case that forwarding needs to be done. We found that even though using atomics to create a non-interleaved block of statements, it can get interleaved. What we did is make the possible recipients active again when a process is about to send them a message. At last, we fixed the problem by saying that only if the uniqueContent is or exceeds the limit of messages MSGLIMIT, then try to go inactive. This way we know when all processes are really done.

6 Verification results

We ran three tests in total, verifying safety and the two LTL formulae. The results are respectively in figures 1, 2 and 3. The run specified it searches at a depth of 1118, where exhaustive verification would be infeasible, we have checked that there are no errors when using bitstate/supertrace.

After checking the manual it states that using this storage option only has minimal side effects.

7 Code

```
#define MAXUSERS        4
#define MAXMESSAGES     2
#define MSGLIMIT            1

/*
 * sender constraint content
 */
chan inbox[MAXUSERS] = [MAXMESSAGES] of { byte, byte, byte
    };
chan outbox[MAXUSERS] = [MAXMESSAGES] of { byte, byte, byte
     };

bit isActive[MAXUSERS];
bit uniqueContent = 0;

/*
 * userDevice reads from the inbox, (might forward ->
    outbox)
 * or sends from the outbox,
 * or creates a new mesage and adds to the outbox,
 * or skip.
 */
proctype userDevice (byte user){
byte inSender;
byte inConstraint;
byte inContent;
byte outSender;
byte outConstraint;
```

```
byte outContent;

do
/* READ AN INBOX MESSAGE */
:: nempty( inbox[ user ] ) ->
   atomic{
   inbox[ user ] ? inSender, inConstraint, inContent;
   d_step{
   isActive[(user)] = 1;
   isActive[(user+1)%MAXUSERS] = 1;
   isActive[(user+2)%MAXUSERS] = 1;
   }
   }
R0: skip;
   atomic{
   if
   :: inConstraint > 1 && user > 0 &&
      (inSender != ((user+1) % MAXUSERS) &&
           inSender != ((user+2) % MAXUSERS)) ->
           if
           :: nfull(outbox[user]) -> outbox[user] !
              inSender, inConstraint, inContent;
           :: full(outbox[user]) -> inbox [ user ] !
              inSender, inConstraint, inContent;
           fi
   :: else -> skip; /*dont send, its constraint is 0*/
   fi;
   }
   /*acknowledge to sender */

/*CREATE A NEW MESSAGE TO OUTBOX */
:: empty(inbox[user]) && empty(outbox[user]) &&
    uniqueContent < MSGLIMIT && user > 0 -> atomic{
   if
   :: outbox[ user ] ! user, 1, uniqueContent; /*outbox
      is not full, so this will not block.*/
   uniqueContent = uniqueContent + 1;
   :: outbox[ user ] ! user, 2, uniqueContent; /*outbox
      is not full, so this will not block.*/
   uniqueContent = uniqueContent + 1;
   fi;
   }
/*SEND AN OUTBOX MESSAGE*/
:: nempty(outbox[user]) ->
   atomic{
   outbox[ user ] ? outSender, outConstraint, outContent;
   d_step{
   isActive[(user)] = 1;
   isActive[(user+1)%MAXUSERS] = 1;
   isActive[(user+2)%MAXUSERS] = 1;
```

```
    }
    }
    if
    :: (nfull(inbox[(user+1) % MAXUSERS]) &&
        nfull(inbox[(user+2) % MAXUSERS])) ->
         inbox[(user+1) % MAXUSERS] ! outSender, (
             outConstraint-1), outContent; /*follower1*/
S0: skip;
    inbox[(user+2) % MAXUSERS] ! outSender, (
        outConstraint-1), outContent; /*follower2*/
S1: skip;
    :: (full(inbox[(user+1) % MAXUSERS]) ||
        full(inbox[(user+2) % MAXUSERS])) -> outbox[ user
            ] ! outSender, outConstraint, outContent;
    fi;
:: uniqueContent >= MSGLIMIT && empty(inbox[user]) &&
    empty(outbox[user]) && isActive[user] == 1 -> isActive
    [user] = 0;
:: uniqueContent >= MSGLIMIT && isActive[user] == 0 && (
    isActive[0] == 1 || isActive[1] == 1 || isActive[2] ==
     1 || isActive[3] == 1) -> skip;
:: uniqueContent >= MSGLIMIT && isActive[0] == 0 &&
    isActive[1] == 0 && isActive[2] == 0 && isActive[3] ==
     0 -> break;
od
}

byte vi = 0;
byte vk = 0;
init {
atomic{
        do
        :: (vi < MAXUSERS) -> vi++;
        :: break;
        od;

        do
        :: (vk < MAXUSERS) -> vk++;
        :: break;
        od;
    }
atomic{
        run userDevice(1);
        run userDevice(2);
        run userDevice(3);
        run userDevice(0);
        }
}

ltl fo { (
```

```
(
([] ((  (vi+1 > 1) &&
        (userDevice[vi+1]@S0) &&
                (userDevice[vi+1]:_3_outConstraint > 1 )
                    &&
                (userDevice[vi+1]:_3_outContent == vk)
->
( <> (  (userDevice[((vi+1)%(MAXUSERS+1))]@R0) &&
        (userDevice[((vi+1)%(MAXUSERS+1))]:_3_inContent
           == vk) )))))
)&&(
([] ((  (vi > 1) && (userDevice[vi]@S1) &&
        (userDevice[vi+1]:_3_outConstraint > 1 ) &&
                (userDevice[vi+1]:_3_outContent == vk)
->
( <> (  (userDevice[((vi+2)%(MAXUSERS+1))]@R0) &&
        (userDevice[((vi+2)%(MAXUSERS+1))]:_3_inContent
           == vk) )))))
)
)
}
ltl fo2 { (
([] (((vi+1 > 0) &&
       userDevice[vi+1]@S0 &&
           (userDevice[vi+1]:_3_outConstraint == 2 ) &&
           (userDevice[vi+1]:_3_outContent == vk) &&
           (((vi+1)%MAXUSERS) > 0)
         ) ->
         (<> ((userDevice[(((vi+1)%MAXUSERS)+1%MAXUSERS)
             ]@R0 &&
                userDevice[(((vi+1)%MAXUSERS)+1%MAXUSERS)
                    ]:_3_inContent == vk) &&
              (userDevice[(((vi+1)%MAXUSERS)+2%MAXUSERS)
                  ]@R0 &&
                userDevice[(((vi+1)%MAXUSERS)+2%MAXUSERS)
                    ]:_3_inContent == vk))
        )))
&&
([] (((vi > 0) &&
       userDevice[vi+1]@S0 &&
           (userDevice[vi+1]:_3_outConstraint == 2 ) &&
           (userDevice[vi+1]:_3_outContent == 2 ) &&
           (((vi+2)%MAXUSERS) > 0)
         ) ->
         (<>((userDevice[(((vi+2)%MAXUSERS)+1%MAXUSERS)]
             @R0 &&
               userDevice[(((vi+2)%MAXUSERS)+1%MAXUSERS)
                   ]:_3_inContent == vk) &&
            (userDevice[(((vi+2)%MAXUSERS)+2%MAXUSERS)]
               @R0 &&
```

```
                userDevice[(((vi+2)%MAXUSERS)+2%MAXUSERS)
                    ]:_3_inContent == vk))
            )))
)}
```

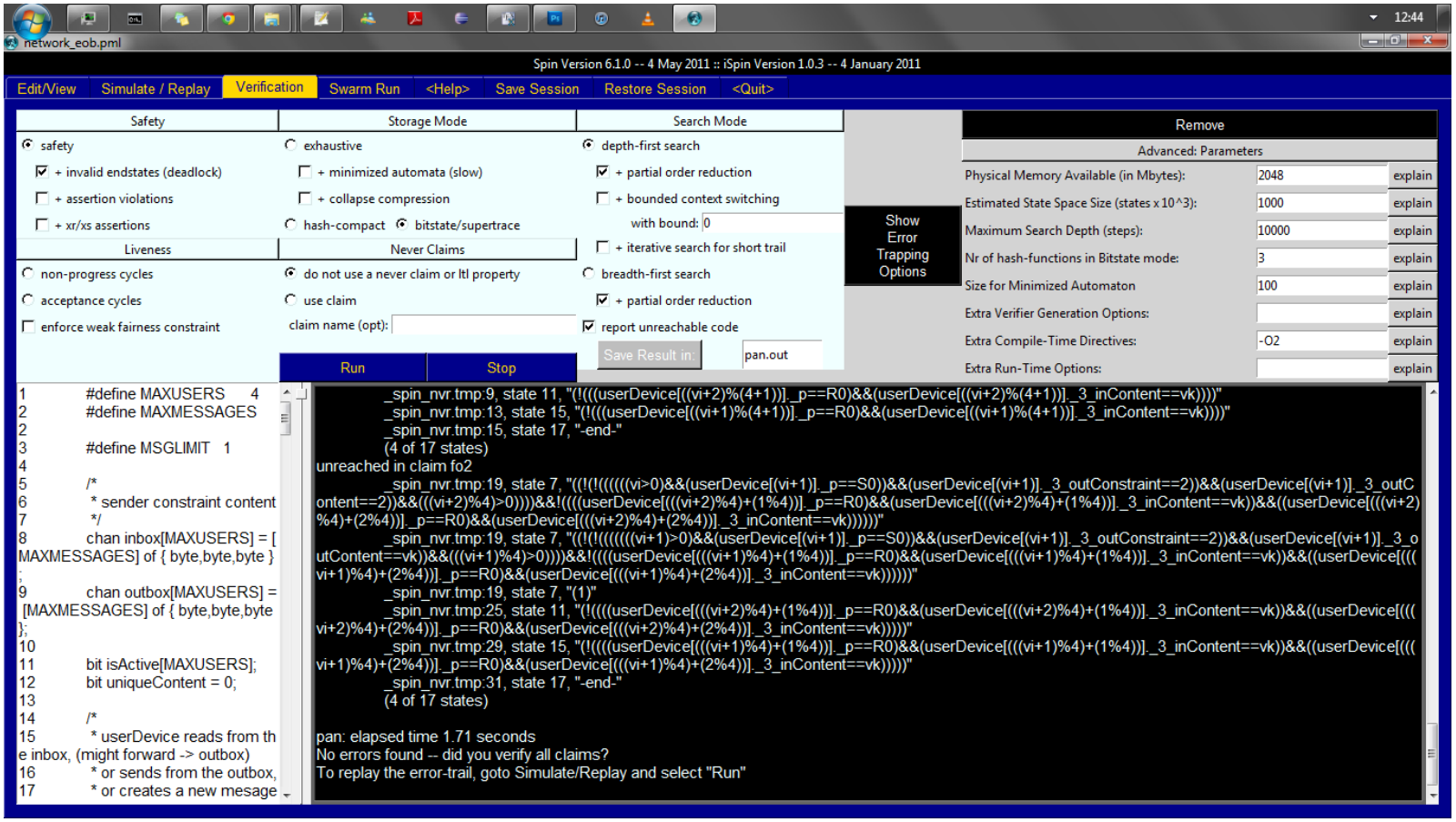

Figure 1: Verification for safety, checking deadlocks

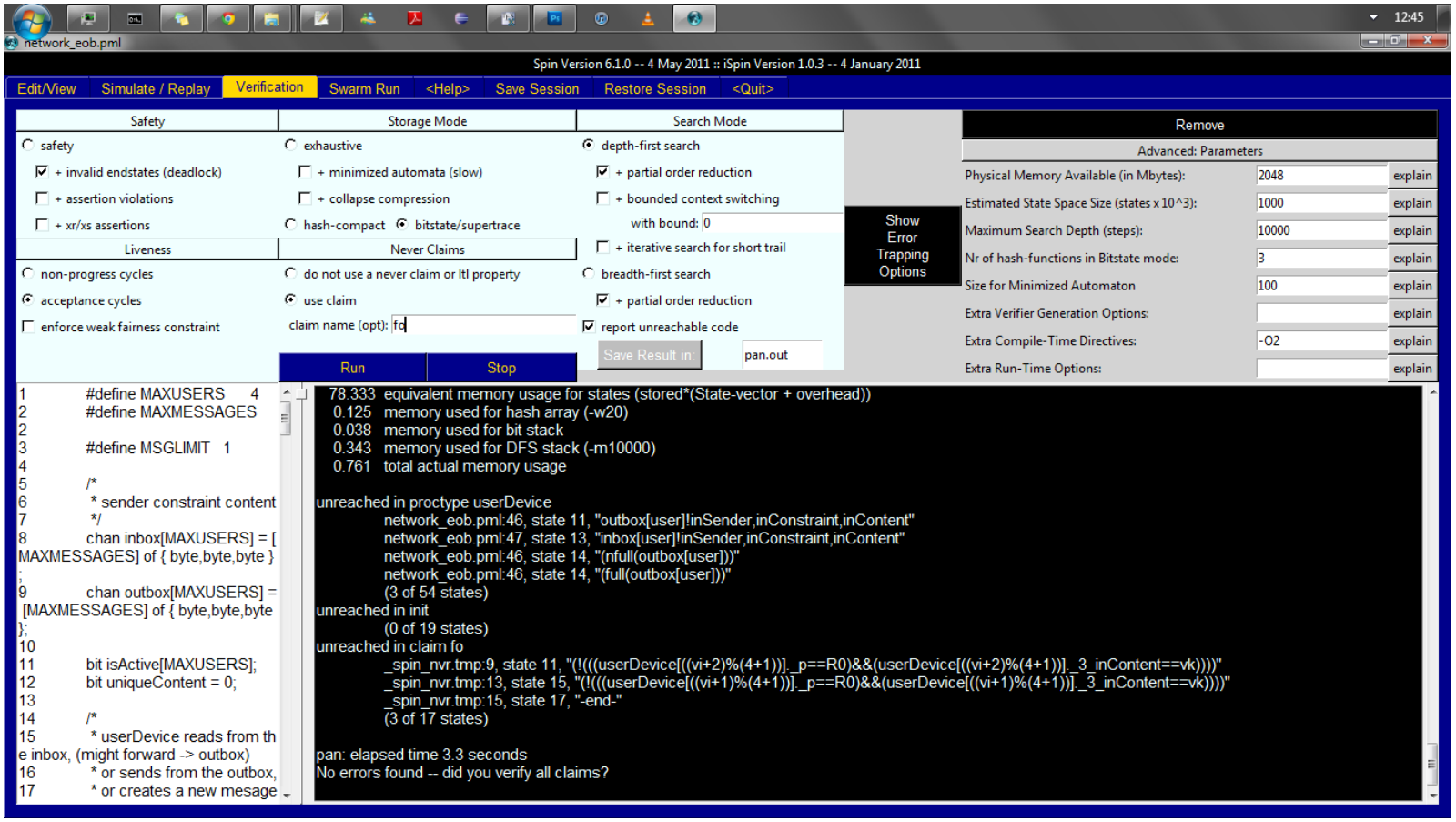

Figure 2: Verification for the Fo claim
11

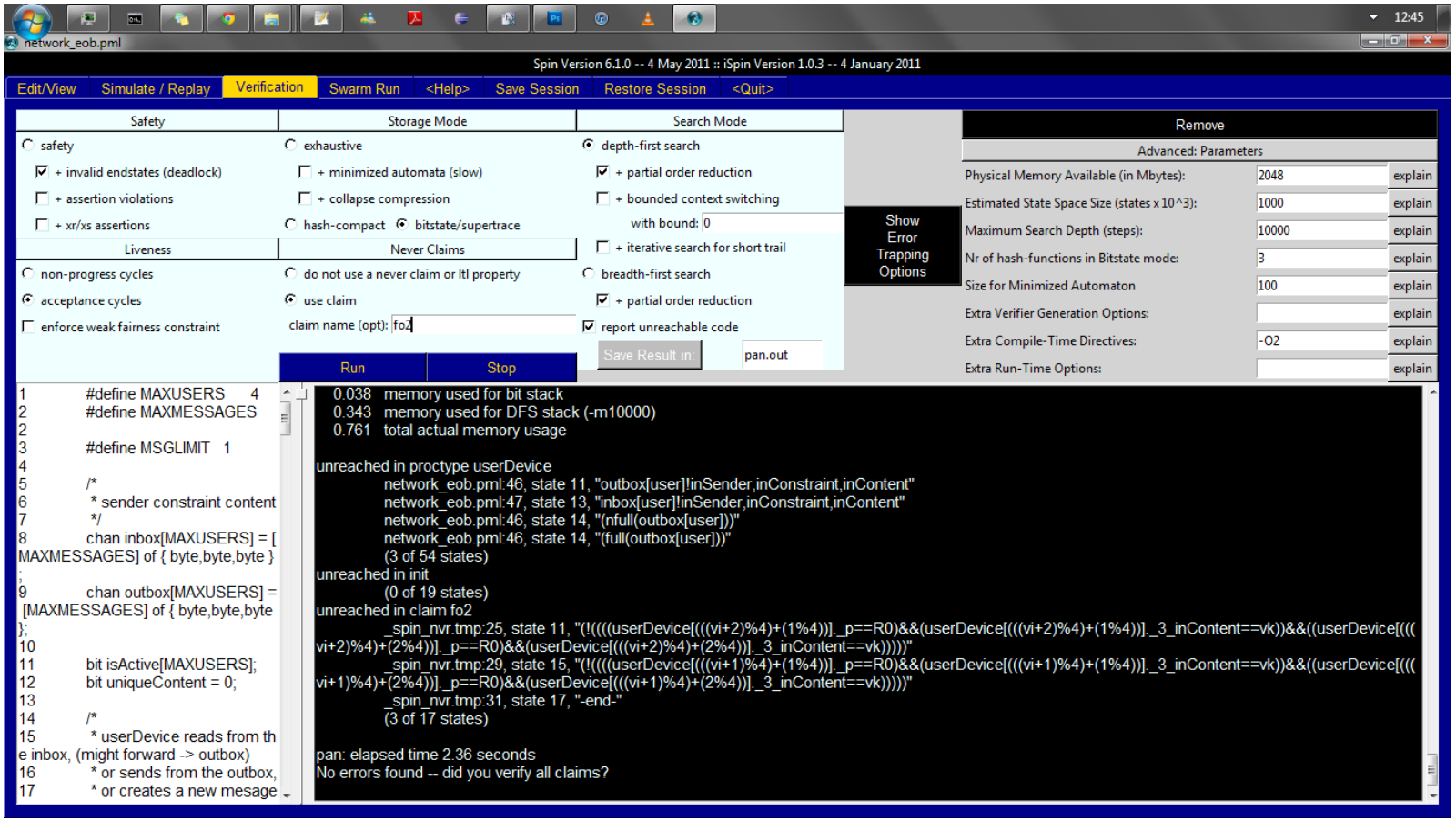

Figure 3: Verification for the Fo2 claim