Formal verification of Anderson mutual exclusion protocol by introducing an auxiliary variable

Naoki Asae, Duong Dinh Tran, and Kazuhiro Ogata
School of Information Science
Japan Advanced Institute of Science and Technology (JAIST)
1-1 Asahidai, Nomi, Ishikawa 923-1292, Japan
Email: {s1910005,duongtd,ogata}@jaist.ac.jp

***Abstract*—The second and third authors of the present paper have formally verified that A-Anderson protocol, which is an abstract version of Anderson mutual exclusion protocol, enjoys the mutual exclusion property in their previous work. The reason why they did not conduct formal verification with the original version of Anderson but with A-Anderson instead is that Anderson uses a finite boolean array and the modulo (or remainder) operation of natural numbers, causing the challenge to conduct formal verification in a sense of theorem proving. Since then, we have successfully completed formal verification with Anderson to which an auxiliary variable is introduced. The protocol is specified in CafeOBJ, an algebraic specification language, and it is formally verified that the protocol enjoys the property with CafeOBJ. The auxiliary variable does not change the behavior of Anderson. We then conclude that Anderson enjoys the mutual exclusion property by proving that the property is an invariant of the specification. We also informally discuss why it is necessary to introduce auxiliary variables so that we can successfully complete formal verification with some protocols or systems.**

***Keywords*-algebraic specification language; mutual exclusion protocol; auxiliary variable; proof score**

I. Introduction

Mutual exclusion is the problem such that at most one thread, process, node, or any execution entity is allowed to enter its critical section to use some shared resources, such as shared memory in concurrent and/or distributed systems. Mechanisms or protocols that solve the problem are called mutual exclusion protocols. Anderson protocol (or Anderson) [1] is a mutual exclusion protocol. Thus, the most important property the protocol should satisfy is the mutual exclusion property. It is, however, challenging to formally verify that Anderson protocol enjoys the mutual exclusion property, in a sense of theorem proving. The reason is that its algorithm uses a finite array and the modulo operation of natural numbers. In the paper [2], the second and third authors of the present paper have introduced an abstract version of Anderson, which is called A-Anderson protocol (or A-Anderson), and formally verified that A-Anderson enjoys the mutual exclusion property.

This research was partially supported by JSPS KAKENHI Grant Number JP19H04082.

DOI reference number: 10.18293/SEKE2021-038

Although as mentioned in the paper [2], the authors have successfully proved that Anderson enjoys the mutual exclusion property by showing that there exists a simulation relation from Anderson to A-Anderson, and such simulation preserves the property, directly proving that Anderson enjoys the property still interests us. In Anderson, each process is located at one of three locations: rs (Remainder Section), ws (Waiting Section), or cs (Critical Section). Initially, each process is located at rs, and when a process wants to enter cs, it first moves to ws from rs. By introducing an auxiliary variable to record a collection of processes currently located at ws or cs, we have successfully completed the proof that Anderson satisfies the mutual exclusion property without using the abstract version A-Anderson but with an auxiliary variable introduced. This modification only records information in the current and past states but does not affect the current or future values of any other variables in the algorithm. Therefore, we can guarantee that adding the auxiliary variable does not change the behavior of Anderson. Originally, we got stuck in the formal verification of Anderson because the proof requires a lemma that is obvious but so tough to prove. Introducing the auxiliary variable helps us to accomplish the proof of that lemma, leading to the complete formal verification. The proof of the lemma has not yet been completed as of the paper submission without introducing any auxiliary variables, though.

Our verification in this paper uses observational transition systems (OTSs) [3] as state machines. The OTS formalizing Anderson is specified in CafeOBJ [4], which is a formal specification language. Then, in the specification, we introduce an auxiliary variable to store a list of processes currently located at ws or cs. Formal proofs are conducted by writing what is called "proof scores" [3] in CafeOBJ and executing them with CafeOBJ. Proof scores are developed by simultaneous structural induction on a state variable of the OTS. We verify that Anderson enjoys the mutual exclusion property by proving that the property is an invariant of the OTS formalizing Anderson. The verification requires the use of some additional lemmas, one of them is the lemma mentioned above that makes us so stuck to prove its correctness.

The rest of the paper is organized as follows: Sect. II describes Anderson protocol. Sect. III presents how to formally specify the protocol in CafeOBJ. Sect. IV presents our proof attempt to formally verify that Anderson protocol enjoys the mutual exclusion property and the reason why we got stuck to complete the proof. Sect. V describes our solution to complete the verification by introducing an auxiliary variable and informally discusses why we need to do so. Some related work is mentioned in Sect. VI. Finally, Sect. VII concludes the paper. The specification and proof scores presented in this paper are available at https://gitlab.com/duongtd23/anderson-au.

II. Anderson Protocol

We suppose that there are N processes participating in Anderson protocol. The pseudo-code of Anderson protocol for each process p can be written as follows:

Loop "Remainder Section"
  rs : $place[p] := \text{fetch\&incmod}(next, N)$;
  ws : **repeat until** $array[place[p]]$;
    "Critical Section"
  cs : $array[place[p]]$,
      $array[(place[p]+1)\,\%\,\text{N}] := \text{false},\ \text{true}$;

We suppose that each process is located at rs, ws or cs and initially located at rs. *place* is an array whose size is N and each of whose elements stores one from $\{0, 1, \ldots, N-1\}$. Initially, each element of *place* can be any from $\{0, 1, \ldots, N-1\}$ but is 0 in this paper. Although *place* is an array, each process p only uses $place[p]$ and then we can regard $place[p]$ as a local variable to each process p. *array* is a Boolean array whose size is N. Initially, $array[0]$ is true and $array[j]$ is false for any $j \in \{1, \ldots, N-1\}$. *next* is a natural number variable and initially set to 0. fetch&incmod$(next, N)$ atomically does the following: setting *next* to $(next+1)\,\%\,N$ and returning the old value of *next*. $x, y := e_1, e_2$ is a concurrent assignment that is processed as follows: calculating e_1 and e_2 independently and setting x and y to their values, respectively.

III. Formal Specification of Anderson Protocol

We use four observation functions `pc`, `next`, `place`, `array` to store information about the location of each process, the value of the global variable *next*, the value stored in each element of *place* and the value stored in each element of *array*, respectively:

```
op pc : Sys Pid -> Label .
op next : Sys -> SNat .
op place : Sys Pid -> SNat .
op array : Sys SNat -> Bool .
```

`Sys` is the sort that represents the state space of Anderson. `Pid` is the sort denoting the set of process IDs. `Label` is the sort that expresses the set of labels (rs, ws and cs). `SNat` is the sort of natural numbers and `Bool` is the sort of Boolean values. Observation function `array` observes the value stored in each element of *array* by passing to `array` the index of element as the second argument.

We also introduce the observer `count` to keep track of the number of processes that would like to enter the Critical Section and/or to be there (i.e., the number of processes currently located at cs or ws):

```
op count : Sys -> SNat .
```

If $N+1$ or more processes participate in the protocol, the protocol does not enjoy the mutual exclusion property, which we realzied when we were formally specifying the protocol in CafeOBJ. This is implicitly assumed by the protocol but it is necessary to make the assumption explicit so as to do formal verification. This is tiny but important, demonstrating worth formally specifying systems.

We have the declaration of N and its property as follows:

```
op N : -> SNzNat .  eq (1 < N) = true .
```

`SNzNat` is the sort of non-zero natural numbers and a subsort on `SNat`. N is expressed as the constant `N` of `SNzNat`. The property says that `N` is greater than `1` because if there is only one process, we do not need to use any mutual exclusion protocols. In the formal specification, we declare `1` as a constant of `SNzNat` that equals `s(0)` (i.e., `1` is successor of `0`).

We use one constructor that represents an arbitrary initial state as follows:

```
op init : -> Sys {constr} .
```

`init` is defined in terms of equations, specifying the values observed by the four observation functions in an arbitrary initial state as follows:

```
eq pc(init,P) = rs .     eq next(init) = 0 .
eq place(init,P) = 0 .   eq count(init) = 0 .
eq array(init,I)
= (if I = 0 then true else false fi) .
```

where `P` is a CafeOBJ variable of `Pid` and `I` is a CafeOBJ variable of `SNat`.

We use three transition functions that are also constructors:

```
op want : Sys Pid -> Sys {constr}
op try  : Sys Pid -> Sys {constr}
op exit : Sys Pid -> Sys {constr}
```

The three transition functions capture the actions that each process moves to ws from rs, tries to move to cs from ws and moves back to rs from cs, respectively. The reachable states are composed of the four constructors.

Each of the three transition functions is defined in terms of equations, specifying how the values observed by the four observation functions change. Let `S` be a CafeOBJ variable of `Sys`, `P` & `Q` be CafeOBJ variables of `Pid` and `I` & `J` be CafeOBJ variables of `SNat`.

`want` is defined as follows:

```
ceq pc(want(S,P),Q)
= (if P = Q then ws else pc(S,Q) fi)
  if c-want(S,P) .
ceq place(want(S,P),Q)
= (if P = Q then next(S) else place(S,Q) fi)
  if c-want(S,P) .
ceq next(want(S,P))
= (s(next(S)) rem N) if c-want(S,P) .
eq array(want(S,P),I) = array(S,I) .
ceq count(want(S,I)) = s(count(S))
  if c-want(S,I) .
ceq want(S,P) = S if c-want(S,P) = false .
```

where `c-want(S,P)` is

```
pc(S,P) = rs and count(S) < N
```

`s` of `s(next(S))` is the successor function of natural numbers. x `rem` y calculates the remainder obtained by dividing x by y. The equations say that if `c-want(S,P)` is true, the location of `P` changes to `ws`, the location of each other process `Q` does not change, the `P`'s *place* changes to *next*, each other process `Q`'s *place* does not change, *next* is updated to $(next + 1)\,\%\,N$, *count* is incremented, and *array* does not change in the state denoted `want(S,P)`; if `c-want(S,P)` is false, nothing changes.

`try` is defined as follows:

```
ceq pc(try(S,P),Q)
= (if P = Q then cs else pc(S,Q) fi)
if c-try(S,P) .
eq place(try(S,P),Q) = place(S,Q) .
eq array(try(S,P)) = array(S) .
eq next(try(S,P),I) = next(S) .
eq count(try(S,I)) = count(S) .
ceq try(S,P) = S if c-try(S,P) = false .
```

where `c-try(S,P)` is

```
pc(S,P) = ws and array(S,place(S,P)) = true
```

The equations say that if `c-try(S,P)` is true, the location of `P` changes to `ws`, the location of each other process `Q` does not change, *place*, *next* and *count* do not change in the state denoted `try(S,P)`; if `c-try(S,P)` is false, nothing changes.

`exit` is defined as follows:

```
ceq pc(exit(S,P),Q)
= (if P = Q then rs else pc(S,Q) fi)
if c-exit(S,P) .
eq place(exit(S,P),Q) = place(S,Q) .
eq next(exit(S,P)) = next(S) .
ceq array(exit(S,P),I) =
(if I = (s(place(S,P)) rem N) then true
 else (if I = place(S,P) then false
 else array(S,I) fi) fi) if c-exit(S,P) .
ceq count(exit(S,I)) = (sd(count(S),1))
  if c-exit(S,I) .
ceq exit(S,P) = S if c-exit(S,P) = false .
```

where `c-exit(S,P)` is `pc(S,P) = cs`. `sd(`x,y`)` returns the difference of x and y. The equations say that if `c-exit(S,P)` is true, the location of `P` changes to `rs`, the location of each other process `Q` does not change, *place* does not change, *next* does not change, *count* is decreased by one, the `I`th element of *array* is set true if `I` equals `s(place(S,P)) rem N`, the `J`th element of *array* is set false if `J` equals `place(S,P)`, and each other element of *array* does not change in the state denoted `exit(S,P)`; if `c-exit(S,P)` is false, nothing changes.

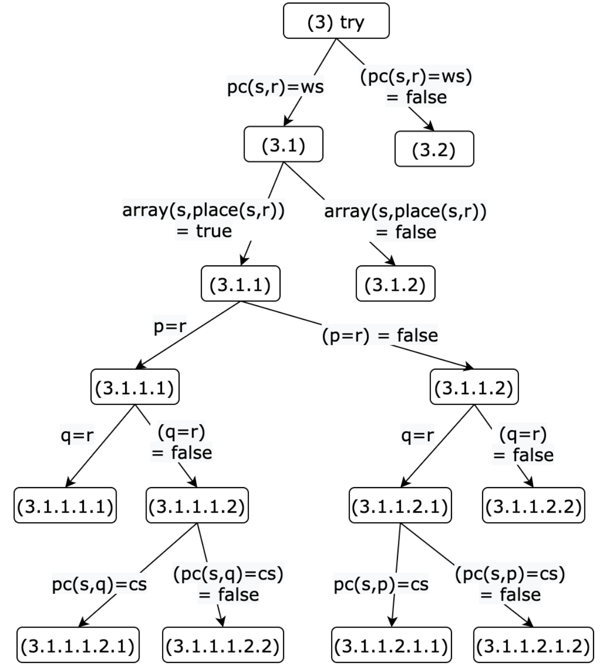

Figure 1. Case spliting for case (3) of the proof of mutex

IV. Formal Verification by Proof Scores

The mutual exclusion property is specified as follows:

```
eq mutex(S,P,Q) = ((pc(S,P) = cs and
  pc(S,Q) = cs) implies (P = Q)) .
```

The equation says that if there are processes in the critical section, there is one, namely that exists at most one process in the critical section at any given moment.

We prove `mutex(S,P,Q)` for all reachable states `S` and all process IDs `P` & `Q` by structural induction on `S`. There are four cases to tackle: (1) `init`, (2) `want`, (3) `try` and (4) `exit`. Let us consider case (3). What to prove is `mutex(try(s,r),p,q)`, where `s` is a fresh constant of `Sys` representing an arbitrary state and `p`, `q` and `r` are fresh constant of `Pid` representing arbitrary process IDs. The induction hypothesis is `mutex(s,P,Q)` for all process IDs `P` & `Q`. Let us note that `s` is shared by `mutex(try(s,r),p,q)` and `mutex(s,P,Q)`, while the variables `P` and `Q` can be replaced with any terms of `Pid`, such as `p` and `q`.

Figure 1 shows the case splitting strategy to prove case (3). Case (3) is first split into two sub-cases: (3.1) `pc(s,r) = ws` and (3.2) `(pc(s,r) = ws) = false`. Case (3.2) can be discharged, its proof score fragment is as follows:

```
open INV .
op s : -> Sys .   ops p q r : -> Pid .
eq (pc(s,r) = ws) = false .
red mutex(s,p,q) implies mutex(try(s,r),p,q) .
close
```

where `INV` is the module in which the specification of Anderson together with `mutex` are available, `open` makes the module `INV` available, `close` stops the use of the module and `red` reduces (computes) the given term. Feeding this proof score fragment into CafeOBJ, CafeOBJ returns true, meaning that the case is discharged.

Case (3.1) is applied case splitting several more times as shown in the Figure. With case (3.1.1.1.2.1), it requires us to use a lemma to discharge the sub-case. Case (3.1.1.1.2.1) says that process `p` is located at `cs`, process `r` (or `q` since `q = r`) is located at `ws` and `array(s,place(s,r)) = true`. In this case, process `r` can move to `cs`, breaking the property concerned because there are two processes `p` and `r` located at `cs`. That is the reason why we need to conjecture a lemma to discharge this case. Such a lemma can be conjectured from the assumptions made in this case. We have conjectured `inv1` as such a lemma, which is as follows:

```
eq inv1(S,P,Q) = ((array(S,place(S,P)) = true
and pc(S,P) = ws and (P = Q) = false) implies
((pc(S,Q) = ws and array(S,place(S,Q)) = true)
or pc(S,Q) = cs) = false) .
```

Then, in the proof score of case (3.1.1.1.2.1), we use `inv1` as a lemma:

```
open INV .
op s : -> Sys .  ops p q r : -> Pid .
eq pc(s, r) = ws .
eq array(s,place(s,r)) = true .
eq p = r .  eq (q = r) = false .
eq pc(s,q) = cs .
red inv1(s,r,q) implies mutex(s, p, q)
implies mutex(try(s, r), p, q) .
close
```

The remaining cases can be discharged likewise. The proof of `mutex` does not require any other lemma except for `inv1`. We need to prove that `inv1` is an invariant of the OTS formalizing Anderson to complete the verification. If `inv1` is not an invariant of the OTS formalizing Anderson, there exists a state such that there are two different processes `P` and `Q` where `P` can freely enter the critical section and `Q` is locating at the critical section or can freely enter the critical section. In both cases, the mutual exclusion property is broken. That is the reason why `inv1` must be an invariant of the OTS formalizing Anderson.

In the proof of `inv1`, we need to use another lemma `inv7` that is as follows:

```
eq inv7(S,P) = (pc(S,P) = ws or pc(S,P) = cs)
implies (0 < count(S)) .
```

where `S` is a CafeOBJ variable of `Sys`, `P` & `Q` are CafeOBJ variables of `Pid`. `inv7` intuitively says that if there exists a process located at `ws` or `cs` in a state `S`, `count(S)` is greater than 0. Considering the roles of `count` that keeps track of the number of processes that have moved to `ws` and not yet left `cs`, namely the number of processes that are located at `ws` or `cs`, `inv7` must be an invariant of the OTS formalizing Anderson.

In the proof of `inv7`, we need to use a lemma `inv7-2` that is as follows:

```
eq inv7-2(S,P,Q) = ((pc(S,P) = ws or pc(S,P)
= cs) and (pc(S,Q) = ws or pc(S,Q) = cs) and
(P = Q) = false) implies (s(0) < count(S)) .
```

`inv7-2` intuitively says that if there are two different processes located at `ws` or `cs` in a state `S`, `count(S)` is greater than 1.

In the proof of `inv7-2`, we need to use a lemma `inv7-3` that is as follows:

```
eq inv7-3(S,P,Q,R) = ((pc(S,P) = ws or pc(S,P)
= cs) and (pc(S,Q) = ws or pc(S,Q) = cs) and
(pc(S,R) = ws or pc(S,R) = cs) and  (P = Q)
= false and (P = R) = false and (Q = R)
= false) implies (s(s(0)) < count(S)) .
```

where `R` is a CafeOBJ variable of `Pid`. `inv7-3` intuitively says that if there are three different processes located at `ws` or `cs` in a state `S`, `count(S)` is greater than 2.

It seems necessary to use an unlimited number of similar lemmas to complete the proof of `inv7`. If CafeOBJ made it possible to use an operator with a variable number of parameters, we could generalize the lemmas:

```
eq inv7-k(S,P1,...,Pk) = ((pc(S,P1) = ws or
pc(S,P1) = cs) and ... and (pc(S,Pk) = ws or
pc(S,Pk) = cs) and (P1 = P2) = false
and ... and (P(k-1) = Pk) = false)
implies (s^{k-1}(0) < count(S)) .
```

where `s^{k-1}(0)` denotes $k-1$. k can vary like a variable and the number k of parameters can change. It is, however, impossible to deal with such an operator with a variable number of parameters in CafeOBJ. That is the reason why we got stuck several months to prove that Anderson enjoys the mutual exclusion property. If we could complete the proof of `inv7`, then the verification is accomplished.

V. Introducing an auxiliary variable

A. *Introducing psInWsCs*

The proof of `inv7` seems so tough despite its obviousness. We overcome the problem by introducing an observer (also can be called an auxiliary variable) `psInWsCs` that records

Table I
CASE SPLITTING FOR THE PROOF OF INV7

(1.1.1.1)	pc(s,p) = cs, csb1, csb2, csb3
(1.1.1.2)	pc(s,p) = cs, csb1, csb2, ¬csb3
(1.1.2)	pc(s,p) = cs, csb1, ¬csb2
(1.2)	pc(s,p) = cs, ¬csb1
(2)	pc(s,p) = rs
(3.1.1.1)	pc(s,p) = ws, csb1, csb2, csb3
(3.1.1.2)	pc(s,p) = ws, csb1, csb2, ¬csb3
(3.1.2)	pc(s,p) = ws, csb1, ¬csb2
(3.2)	pc(s,p) = ws, ¬csb1

all processes currently located at ws or cs. The observer is declared as follows:

```
op psInWsCs : Sys -> SetPids .
```

where `SetPids` is the sort denoting the set of process IDs. `psInWsCs` is defined in the initial states and each transition as follows:

```
eq psInWsCs(init) = emp .
ceq psInWsCs(want(S,P)) =
  insert(P,psInWsCs(S)) if c-want(S,P) .
eq psInWsCs(try(S,P)) = psInWsCs(S) .
ceq psInWsCs(exit(S,P)) =
  delete(P,psInWsCs(S)) if c-exit(S,P) .
```

where `emp` is the constant of sort `SetPids` representing the empty set. The equations say that initially, `psInWsCs` is empty; when process `P` moves to ws from rs, `P` is inserted into `psInWsCs`; when process `P` moves to rs from cs, `P` is removed from `psInWsCs`; `psInWsCs` does not change when `P` moves to cs from ws.

Then, we can complete the proof of `inv7` by using the following lemmas:

```
eq inv10(S) = #(psInWsCs(S)) = count(S) .
eq inv11(S,P) = (pc(S,P) = ws or pc(S,P) = cs)
 implies P \in psInWsCs(S) .
eq inv12(SE,E) = E \in SE implies 0 < #(SE) .
```

where `#` is the operator taking a set as the parameter and returning the size of it, `\in` is the infix operator checking the existence of an element in a set, `SE` and `E` are CafeOBJ variables denoting arbitray set and element, respectively. To prove `inv7`, we do not need to apply structural induction, but only case splitting is enough. Table I shows the case splitting for the proof of `inv7`, where each csb$i$ for $i = 1, 2, 3$ is as follows:

csb1 ≜ `(count(s) = #(psInWsCs(s)))`
csb2 ≜ `p \in psInWsCs(s)`
csb3 ≜ `0 < #(psInWsCs(s))`

For example, the proof fragment of case (1.1.1.2) is as follows:

```
open INV .
ops p r : -> Pid . op s : -> Sys .
eq pc(s,p) = cs .
eq count(s) = #(psInWsCs(s)) .
eq p \in psInWsCs(s) = true .
eq (0 < #(psInWsCs(s))) = false .
red inv12(psInWsCs(s),p) implies inv7(s,p) .
close
```

The proof of this case uses `inv12` as a lemma. Let us repeat again that, to prove `inv7`, we do not apply structural induction, but only conduct case splitting. The proof of case (3.1.1.2) also uses `inv12` as a lemma. The proofs of cases (1.1.2) and (3.1.2) use `inv11` as a lemma. The proofs of cases (1.2) and (3.2) use `inv10` as a lemma. The remaining cases are proved without any lemma.

To complete the verification, we also use the following lemmas:

```
eq inv2(S,P) = ((pc(S,P) = cs)
 implies (array(S,place(S,P)) = true)) .
eq inv3(S,G,H) = (((G = H) = false and
 array(S,G) = true)
 implies (array(S,H) = false)) .
eq inv4(S,P,Q) = (place(S,P) = place(S,Q) and
 (pc(S,P) = rs) = false and (P = Q) = false)
 implies (pc(S,Q) = rs) .
eq inv5(S,P,I) = (pc(S,P) = ws and
 place(S,P) = ((I + next(S)) rem No))
 implies ((I + count(S)) < No) = false .
eq inv6(S,P) = (place(S,P) < No) .
eq inv8(S,I) = (array(S,I) = true implies
 next(S) = (I + count(S)) rem No) .
eq inv9(S) = (next(S) < No) .
```

The proof of `inv1` uses `mutex`, `inv4`, `inv8`, and `inv7` as lemmas. The proof of `inv2` uses `mutex` as a lemma. The proof of `inv3` requires the use of `inv2` as a lemma. `inv4` would be the most complicated invariant, its proof uses `inv1`, `inv2`, `inv5`, `inv6`, `inv7`, and `inv8` as lemmas. To prove `inv5`, we need to use `inv2`, `inv4`, `inv6`, `inv7`, and `inv8` as lemmas. `mutex` and `inv9` are used as a lemma in the proof of `inv6`. The proof of `inv8` uses `inv2` and `inv3` as lemmas. We can prove `inv9`, `inv11`, and `inv12` without using any other lemma. The proof of `inv10` requires to use `inv11` as a lemma.

B. Discussion

To prove a property is an invariant of an OTS, we need to conjecture some additional lemmas that are also invariants on the fly during the proof. It is often the case such that the lemma conjectured is not easy to prove such as `inv7` in this paper. Sometimes, in some non-trivial sub-cases of the induction proof, we do not have enough information to verify that the lemma is preserved by a transition. Let us return to `inv7` and its lemmas `inv7-2`, `inv7-3`, etc. in the last section to explain the difficulty made us could not complete the proofs of them and the reason why it is necessary to introduce auxiliary variables like `psInWsCs`.

The premise of `inv7` or each `inv7-k` says that there exists a set of processes that currently located at ws or cs, and its corresponding conclusion concludes that `count` is greater than or equal to the size of that set. However, we do not have enough information to calculate the value of `count` to make the comparison because we can not observe the full set of all processes currently located at ws or cs. We only know that there explicitly is/are one (`P`), or two (`P` and `Q`), or three (`P`, `Q`, and `R`) process(es) currently located at `ws` or `cs` corresponding to each `inv7`, or `inv7-2`, or `inv7-3`, respectively. That is the reason why the proof of `inv7` or each of `inv7-k` becomes so tough or even almost impossible. Since the difficulty comes from the impossibility of observing the full set of all processes currently located at ws or cs, we introduce the observer `psInWsCs` recording the collection of processes that have entered ws and not yet left cs. Consequently, we overcome the difficulty, accomplish the proof `inv7` as well as the complete verification.

VI. Related Work

Tran and Ogata [2] have made an abstract version of Anderson, which is called A-Anderson protocol, and formally verified that A-Anderson enjoys the mutual exclusion property. The verification is conducted in three ways: (1) by writing proof scores in CafeOBJ, (2) with a proof assistant CiMPA [5] for CafeOBJ and (3) with a proof generator CiMPG [5] for CafeOBJ. The paper has also mentioned how to formally verify that Anderson enjoys the mutual exclusion property by showing that there exists a simulation relation from Anderson to A-Anderson, and such simulation preserves the property. The details of this verification technique, however, were not presented in [2] due to the page-limitation. They mentioned that they would report that part in a longer version.

Lamport and Merz [6] has described how to introduce auxiliary variables into TLA+ specifications to prove a refinement mapping between two TLA+ specifications (i.e., the set of observable behaviors of the first specification is a subset of the behaviors of the second one). Auxiliary variables have been classified into three kinds: history, prophecy and stuttering variables. History variables are used to record what has happened in the past (including the present). Prophecy variables are used to predict what will happen in the future. Stuttering variables are used to introduce stuttering steps. `psInWsCs` we have used in this paper corresponds to a history variable. Lamport and Merz use auxiliary variables to make it possible to find a refinement map from a TLA+ specification to another TLA+ specification. While we use `psInWsCs` to complete the proof that a property is an invariant of the OTS formalizing Anderson.

Auxiliary variables, go back to the past were originally introduced by Owicki and Gries [7] in the form of history variables. Later, Abadi and Lamport [8] have introduced the idea of prophecy variables. In [8], Abadi and Lamport have presented how to use both history and prophecy variables to prove that one program is a correct implementation of a specification, by showing that the former refines the latter.

VII. Conclusion

We have formally verified that Anderson protocol to which an auxiliary variable is introduced enjoys the mutual exclusion property. Consequently, we can conclude that Anderson enjoys the property. Originally, we got stuck several months in the verification attempt because the proof requires a lemma that is so tough to prove the correctness of it. Introducing an auxiliary variable `psInWsCs` helps us to accomplish the proof of that lemma, leading to the complete formal verification. `psInWsCs` records all processes currently located at cs or ws, which means that it does not affect the current or future values of any other variables. Thus, it can be guaranteed that adding `psInWsCs` does not change the behavior of Anderson.

Conjecture lemma has been considering as one of the most challenging tasks to formally prove that a property is an invariant of an OTS. Normally, we can not always conjecture the best lemma every time we need to use a lemma. Sometimes, the lemma is so tough or even almost impossible to prove such as `inv7` in this paper. Then, introducing auxiliary variables into the specification can help us to complete the lemma's proof as well as the formal verification such as `psInWsCs` in the present paper. We can understand the reason why we need to introduce `psInWsCs` in the formal verification of Anderson case study. However, in general, we have not had a contented answer for the question: when we need to introduce auxiliary variables to complete formal verification of other case studies? That should be one piece of our future work to answer such a question.

References

[1] T. E. Anderson, "The performance of spin lock alternatives for shared-memory multiprocessors," *IEEE Trans. Parallel Distrib. Syst.*, vol. 1, no. 1, pp. 6–16, 1990.

[2] D. D. Tran and K. Ogata, "Formal verification of an abstract version of Anderson protocol with CafeOBJ, CiMPA and CiMPG," in *SEKE 2020*, 2020, pp. 287–292.

[3] K. Ogata and K. Futatsugi, "Proof scores in the OTS/CafeOBJ method," in *FMOODS 2003*, 2003, pp. 170–184.

[4] R. Diaconescu and K. Futatsugi, *CafeOBJ Report*, ser. AMAST Series in Computing. World Scientific, 1998, vol. 6.

[5] A. Riesco and K. Ogata, "Prove it! Inferring formal proof scripts from CafeOBJ proof scores," *ACM Trans. Softw. Eng. Methodol.*, vol. 27, no. 2, pp. 6:1–6:32, 2018.

[6] L. Lamport and S. Merz, "Auxiliary variables in TLA+," *CoRR*, vol. abs/1703.05121, 2017.

[7] S. S. Owicki and D. Gries, "An axiomatic proof technique for parallel programs I," *Acta Informatica*, vol. 6, pp. 319–340, 1976.

[8] M. Abadi and L. Lamport, "The existence of refinement mappings," *Theor. Comput. Sci.*, vol. 82, no. 2, pp. 253–284, 1991.