Formal verification of an abstract version of Anderson protocol with CafeOBJ, CiMPA and CiMPG

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

***Abstract*— Anderson protocol is a mutual exclusion protocol. It uses a finite Boolean array shared by all processes and the modulo (or reminder) operation of natural numbers. This is why it is challenging to formally verify that the protocol enjoys the mutual exclusion property in a sense of theorem proving. Then, we make an abstract version of the protocol called A-Anderson protocol that uses an infinite Boolean array instead. We describe how to formally specify A-Anderson protocol in CafeOBJ, an algebraic specification language and how to formally verify that the protocol enjoys the mutual exclusion property in three ways: (1) by writing proof scores in CafeOBJ, (2) with a proof assistant CiMPA for CafeOBJ and (3) with a proof generator CiMPG for CafeOBJ. We mention how to formally verify that Anderson protocol enjoys the property by showing that A-Anderson protocol simulates Anderson protocol.**

***Keywords*-algebraic specification language; mutual exclusion protocol; proof assistant; proof generator; proof score**

I. Introduction

Anderson protocol [1] is a mutual exclusion protocol. The protocol uses a finite Boolean array whose size is the same as the number of processes participating in the protocol. It also uses the modulo operation of natural numbers and an atomic operation $\mathrm{fetch\&incmod}$. $\mathrm{fetch\&incmod}$ takes a natural number variable x and a non-zero natural number constant N and atomically does the following: setting x to $(x+1)\,\%\,\mathrm{N}$, where $\%$ is the modulo operation, and returning the old value of x.

It is challenging to formally verify that Anderson protocol satisfies desired properties, such as the mutual exclusion property, in a sense of theorem proving. This is because the protocol uses a finite array and the modulo operation of natural numbers. Then, we make an abstract version of the protocol by using an infinite Boolean array instead of a finite Boolean array, using $\mathrm{fetch\&inc}$ instead of $\mathrm{fetch\&incmod}$ and stopping use of the modulo operation, where $\mathrm{fetch\&inc}$ is an atomic operation that atomically does the following: setting x to $x + 1$ and returning the old value of x. The abstract version is called A-Anderson protocol or A-Anderson. A-Anderson is formalized as an observation transition system (OTS) [2], [3], the OTS is specified in CafeOBJ [4] and it is formally verified in three ways that the OTS enjoys the mutual exclusion property with CafeOBJ, CiMPA [5] and CiMPG [5]. CafeOBJ is an algebraic specification language. Its processor is called CafeOBJ. The first implementation of CafeOBJ was done in Common Lisp, while the second implementation was done in Maude [6], a sibling language of CafeOBJ. The second implementation is called CafeInMaude [7]. CafeInMaude Proof Assistant (CiMPA) is a proof assistant for CafeOBJ and CafeInMaude Proof Generator (CiMPG) is a proof generator that takes annotated proof scores in CafeOBJ and generates proof scripts for CiMPA.

Proof scores can be written in a similar way to write programs in a similar sense of Larch Prover (LP) [8]. The proof score approach to formal verification is flexible in this sense. This is one advantage of the approach. The approach, however, has a disadvantage. Proof scores are subject to human errors. What CafeOBJ essentially does for proof scores is reduction. If human users overlook some cases, CafeOBJ does not point them out. To get rid of the disadvantage, CiMPA has been developed. Although CiMPA is not subject to human errors, it is not flexible enough. To make each advantage of proof score and CiMPA available, CiMPG has been developed. Given proof scores that should be annotated a little bit, CiMPG generates proof scripts that are fed into CiMPA. If CiMPA can successfully discharge all goals with the generated proof scripts, the proof scores are correct for the goals.

The rest of the paper is organized as follows: Sect. II mentions Anderson protocol and its abstract version. Sect. III describes the formal specification of the abstract version in CafeOBJ. Sect. IV describes the formal verification that the abstract version enjoys the mutual exclusion property by writing proof scores in CafeOBJ. Sect. V describes the formal verification with CiMPA. Sect. VI describes the formal verification with CiMPG. Sect. VII mentions simulation-based verification between A-Anderson and Anderson protocols. Sect. VIII mentions related work. Sect. IX concludes the paper.

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

DOI reference number: 10.18293/SEKE2020-064

II. Anderson Protocol and its Abstract Version

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

```
Loop "Remainder Section"
    rs : $place[i] :=$ fetch&incmod($next, N$);
    ws : repeat until $array[place[i]]$;
        "Critical Section"
    cs : $array[place[i]]$,
            $array[(place[i] + 1) \% \mathrm{N}] :=$ false, 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 i only uses $place[i]$ and then we can regard $place[i]$ as a local variable to each process i. *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.

We also suppose that there are N processes participating in an abstract version of Anderson protocol. The abstract version is called A-Anderson protocol. The pseudo-code of A-Anderson protocol for each process i can be written as follows:

```
Loop "Remainder Section"
    rs : $place[i] :=$ fetch&inc($next$);
    ws : repeat until $array[place[i]]$;
        "Critical Section"
    cs : $array[place[i] + 1] :=$ true;
```

We use an infinite Boolean array *array* instead of a finite one and do not use %. fetch&inc is used instead of fetch&incmod. fetch&inc(*next*) atomically does the following: setting *next* to $next + 1$ and returning the old value of *next*. We also suppose that each process is located at rs, ws or cs and initially located at rs. Initially, each element of *place* can be any natural number but is 0 in this paper, $array[0]$ is true, $array[j]$ is false for any non-zero natural number j and *next* is 0.

III. Specification of A-Anderson Protocol

Each state of A-Anderson protocol can be characterized by the following pieces of information: the location of each process, the value stored in *next*, the value stored in each element of *place* and the value stored in each element of *array*. Therefore, we use the following observation functions:

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

where `Sys` is the sort of states, `Pid` is the sort of process IDs, `Label` is the sort of rs, ws and cs, `SNat` is the sort of natural numbers and `Bool` is the sort of Boolean values. We do not use any infinite arrays in the specification. Instead, we use the observation function `array` to observe the value stored in each element that is given to `array` as its second argument.

We use one constructor that represents an arbitrary initial state:

```
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 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)) if c-want(S,P) .
eq array(want(S,P),I) = array(S,I) .
ceq want(S,P) = S if c-want(S,P) = false .
```

where `c-want(S,P)` is `pc(S,P) = rs`. `s` of `s(next(S))` is the successor function of natural numbers. 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 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) .
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* does not change, *array* does not change and *next* does 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)) then true
 else array(S,I) fi) if c-exit(S,P) .
ceq exit(S,P) = S if c-exit(S,P) = false .
```

where c-exit(S,P) is pc(S,P) = cs. 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, the Ith element of *array* is set true if I equals s(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.

IV. Formal Verification with Proof Scores

Let S be a CafeOBJ variable of Sys, P & Q be CafeOBJ variables of Pid and I & J be CafeOBJ variables of SNat. One desired property A-Anderson protocol should satisfy is the mutual exclusion property that is expressed as follows:

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

The expression (or the term) 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.

To prove that A-Anderson protocol enjoys the property, we need to use the following lemmas:

```
eq inv1(S,P,Q)
= ((pc(S,P) = ws and array(S,place(S,P))
   = true and (P = Q) = false)
  implies
  (pc(S,Q) = cs or (pc(S,Q) = ws and
   array(S,place(S,Q)) = true)) = false) .
eq inv2(S,P)
= ((pc(S,P) = cs)
  implies (array(S,place(S,P)) = true)) .
eq inv3(S,P,Q)
= ((place(S,P) = place(S,Q) and (P = Q)
   = false)
  implies (place(S,P) = 0)) .
eq inv4(S,P)
= (place(S,P) = next(S)
  implies (next(S) = 0)) .
eq inv5(S,P)
= (place(S,P) < s(next(S))) = true .
eq inv6(S,P)
= (pc(S,P) = cs or (pc(S,P) = ws and
   array(S,place(S,P)) = true))
  implies array(S,next(S)) = false .
eq inv7(S) = array(S,s(next(S))) = false .
eq inv8(S,I,J)
= (array(S,J) = true and I < s(J))
  implies array(S,I) = true .
```

where s used in s(next(S)) and s(J) is the successor function of natural numbers.

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.

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, while it is necessary to split case (3.1) into two sub-cases: (3.1.1) q = r and (3.1.2) (q = r) = false. It is also necessary to split case (3.1.1) into two sub-cases: (3.1.1.1) p = r and (3.1.1.2) (p = r) = false. Case (3.1.1.1) can be discharged, while it is still necessary to split (3.1.1.2) into two sub-cases: (3.1.1.2.1) array(s,place(s,r)) = true and (3.1.1.2.2) array(s,place(s,r)) = false. Case (3.1.1.2.2) can be discharged, but we need to split case (3.1.1.2.1) into two sub-cases again: (3.1.1.2.1.1) pc(s,p) = cs and (3.1.1.2.1.2) (pc(s,p) = cs) = false. Feeding the proof scores of case (3.1.1.2.1.1) and case (3.1.1.2.1.2) into CafeOBJ, CafeOBJ returns false and true, respectively. Case (3.1.1.2.1.1) 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 case

(3.1.1.2.1.1), process `r` can move to `cs`, breaking the property concerned because there are two processes `p` and `r` located at `cs`. Therefore, we need to conjecture a lemma to discharge case (3.1.1.2.1.1). Such a lemma can be conjectured from the assumptions made in case (3.1.1.2.1.1). We have conjectured `inv1` as such a lemma. The proof score of case (3.1.1.2.1.1) is as follows:

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

In order to discharge case (3.1.2), we need to split it into two sub-cases: (3.1.2.1) `p = r` and (3.1.2.2) `(p = r) = false`. If `p` and `q` are swapped, case (3.1.2.1) becomes exactly the same as case (3.1.1.2). Hence, case (3.1.2.1) can be discharged in the same way as case (3.1.1.2). We also need to use `inv1` as a lemma but should use `inv1(s,r,q)` instead of `inv1(s,r,p)`. The proof score of a sub-case derived from case (3.1.2.1) that corresponds to case (3.1.1.2.1.1) is as follows:

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

(3.1.2.2) is the only unresolved sub-case of case (3). Once again, this case is split into two sub-cases: (3.1.2.2.1) `p = q` and (3.1.2.2.2) `(p = q) = false`. The former can be discharged, while we need to split the latter into two sub-cases: (3.1.2.2.2.1) `array(s,place(s,r)) = true` and (3.1.2.2.2.2) `array(s,place(s,r)) = false`. Both cases can be discharged. Then, case (3) has been discharged.

Case (4) can be discharged in a similar way as case (3) is discharged. We can discharge case (2) without using any lemmas. It is straightforward to discharge case (1). We need to prove `inv1` to complete the formal verification. The proof of `inv1` uses `inv2`, `inv3`, `mutex` and `inv6` as lemmas. `inv2` and `inv5` can be proved independently without use of any other lemmas. The proof of `inv3` uses `inv4` as a lemma. The proof of `inv4` uses `inv5` as a lemma. The proof of `inv6` uses `inv1`, `inv4`, `mutex` and `inv7` as lemmas. The proof of `inv7` uses `inv2`, `inv6` and `inv8` as lemmas. The proof of `inv8` uses `inv2` as a lemma. Let us note that although the proof of `mutex` uses `inv1` as a lemma and the proof of `inv1` uses `mutex` as a lemma, our argument is not circular. We use simultaneous induction to conduct our proof.

To prove each invariant for an OTS by writing proof scores in CafeOBJ, we first use simultaneous induction on states and do the following: for the base case, it is usually straightforward to discharge the case, and for each induction case, we conduct case splittings and use instances of induction hypotheses (or lemmas) as premises of implications.

It took much less than 1s to run all proof scores with CafeOBJ so as to formally verify that A-Anderson protocol enjoys the mutual exclusion property. The experiment used a computer that carried 3.4GHz microprocessor and 32GB main memory. The same computer was used to conduct the other experiments mentioned in the present paper.

V. Formal Verification with CiMPA

The proof score approach to formal verification does not require to explicitly construct proof trees. The outcomes of the approach are open-close fragments written in CafeOBJ that correspond to leaf parts of proof trees. Conducing formal verification by writing proof scores in CafeOBJ, however, we implicitly construct proof trees. Once we have completed formal verification by writing proof scores in CafeOBJ, we must be able to conduct the formal verification with CiMPA. We partially describe formal verification with CiMPA that A-Anderson enjoys the mutual exclusion property.

We first introduce the goals to prove for CiMPA with the command `:goal` as follows:

```
open INV .
:goal{
  eq [inv1 :nonexec]
    : inv1(S:Sys,P:Pid,Q:Pid) = true .
  eq [inv2 :nonexec]
    : inv2(S:Sys,P:Pid) = true .
  ...
  eq [mutex :nonexec]
    : mutex(S:Sys,P:Pid,Q:Pid) = true .
}
```

where the six more lemmas should be written in the place `...`, `inv1`, `inv2` and `mutex` written in square brackets are the names referring to the goals, respectively, and `:nonexec` instructs CafeOBJ not to use the equations as rewrite rules.

Then, we select `S` with the command `:ind on` as the variable on which we start proving the goals by simultaneous induction:

```
:ind on (S:Sys)
:apply(si)
```

The command `:apply(si)` starts the proof by simultaneous induction on `S`, generating four sub-goals for `exit`, `init`, `try` and `want`, where `si` stands for simultaneous induction. Each sub-goals consists of nine equations to prove. We skip the sequence of commands that discharge the first two sub-goals for `exit` and `init`. We partially describe how to discharge the third sub-goal for `try`. To this end, the first command used is as follows:

```
:apply(tc)
```

where `tc` stands for theorem of constants. The command generates nine sub-goals, one of which is as follows:

```
3-9. TC  eq [mutex :nonexec]:
mutex(try(S#Sys,P#Pid),P@Pid,Q@Pid) = true .
```

The command `:apply(tc)` replaces CafeOBJ variables with fresh constants in goals. `S#Sys` and `P#Pid` are fresh constants introduced by `:apply(si)`, while `P@Pid` and `Q@SNat` are fresh constants introduced by `:apply(tc)`.

To discharge goal 3-9, the following commands are first introduced:

```
:def csb3_9_1 =
 :ctf {eq pc(S#Sys,P#Pid) = ws .}
:apply(csb3_9_1)
:def csb3_9_2 = :ctf {eq Q@Pid = P#Pid .}
:apply(csb3_9_2)
:def csb3_9_3 = :ctf {eq P@Pid = P#Pid .}
:apply(csb3_9_3)
```

Case splittings are carried out based on these three equations. For one generated sub-goal in which we assume that the three equations hold, we use the following commands:

```
:imp [mutex] by
 {P:Pid <- P@Pid ; Q:Pid <- Q@Pid ;}
:apply (rd)
```

The induction hypothesis is instantiated by replacing the variables `P:Pid` and `Q:Pid` with the fresh constants `P@Pid` and `Q@Pid` and the instance is used as the premise of the implication. Then, `:apply(rd)` is used to check if the current goal can be discharged. The goal is discharged in this case. The goal corresponds to case (3.1.1.1) in the last section.

After that, the following commands are written:

```
:def csb3_9_4 =
 :ctf [ array(S#Sys,place(S#Sys,P#Pid)) .]
:apply(csb3_9_4)
:def csb3_9_5 =
 :ctf {eq pc(S#Sys,P@Pid) = cs .}
:apply(csb3_9_5)
```

Case splittings are carried out based on one Boolean term and one equation. For one generated sub-goal in which we assume that the Boolean term is true and the equation holds, we use the following commands:

```
:imp [inv1] by
 {P:Pid <- P#Pid ; Q:Pid <- P@Pid ;}
:imp [mutex] by
 {P:Pid <- P@Pid ; Q:Pid <- Q@Pid ;}
:apply (rd)
```

The lemma `inv1` is instantiated by replacing the variables `P:Pid` and `Q:Pid` with the fresh constants `P#Pid` and `Q#Pid` and the instance is used as the premise of the implication. Next, the induction hypothesis is instantiated by replacing the variables `P:Pid` and `Q:Pid` with the fresh constants `P@Pid` and `Q@Pid` and the instance is used as the premise of the implication. Then, `:apply(rd)` is used to check if the current goal can be discharged. The goal is discharged in this case. The goal corresponds to case (3.1.1.2.1.1) in the last section.

When CiMPA is used to formally verify invariant properties for an OTS, what to do is essentially the same as we do formal verification by writing proof scores in CafeOBJ. The difference is as follows: it is necessary to use the commands given by CiMPA when CiMPA is used.

It took about 22s to run the proof scripts with CiMPA so as to formally verify that A-Anderson protocol enjoys the mutual exclusion property.

VI. Formal Verification with CiMPG

After writing proof scores that A-Anderson protocol enjoys the mutual exclusion property, we can confirm that the proof scores are correct by doing the formal verification with CiMPA as described in the last section. Although we are able to conduct the formal verification with CiMPA once we have completed formal verification by writing proof scores in CafeOBJ, it would be preferable to automatically confirm the correctness of proof scores. CiMPG makes it possible to automatically confirm the correctness of proof scores by generating proof scripts for CiMPA from the proof scores.

To use CiMPG, we need to add one open-close fragment to the proof scores. The open-close fragment is as follows:

```
open INV .
  :proof(ander)
close
```

Moreover, we need to write `:id(ander)` in each open-close fragment. For example, the first open-close fragment used in Sect. IV becomes as follows:

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

Feeding the annotated proof scores into CiMPG, CiMPG generates the proof script for CiMPA. The generated proof script is quite similar to the one written manually. Feeding the generated proof script into CiMPA, CiMPA discharges all goals, confirming that the proof scores are correct. It took about 626s to generate the proof script with CiMPG.

VII. A-Anderson Protocol Simulates Anderson Protocol

We can use "simulation-based verification for invariant properties [9]" so as to formally verify that Anderson protocol enjoys the mutual exclusion property. To this end, we first need to prove that the OTS formalizing A-Anderson protocol simulates the OTS formalizing Anderson protocol by showing that there exists a simulation relation from the latter OTS to the former OTS. We next need to prove that the simulation relation preserves the mutual exclusion property. Then, since we have formally verified that A-Anderson protocol enjoys the property, we can conclude that Anderson protocol also enjoys the property. We will describe this part in a longer version of the present paper.

VIII. Related Work

Anderson protocol has been formally specified in CafeOBJ and semi-formally verified with CafeOBJ [10]. Proof scores have been partially written and then all necessary lemmas have not been conjectured and used. They have used a simulation relation between Ticket protocol and Anderson protocol, where the former is abstract, while the latter is concrete. But, they have not used any precise definitions of simulation relations.

In the paper [9] that proposes simulation-based verification for invariant properties in the OTS/CafeOBJ method, Alternating Bit Protocol (ABP), a communication protocol, is used as an example. Two more abstract protocols are used. The paper concludes that it is not very beneficial to use the simulation-based verification technique in order to formally verify that ABP enjoys desired invariant properties. It is useful to use the technique so as to formally verify that Anderson protocol enjoys the mutual exclusion property, however, although the present paper does not describe the part in detail.

Farn Wang [11] proves that it is impossible to automatically formally verify that concurrent software systems as processes running algorithms on data-structures with pointers enjoy desired properties if there are an arbitrary number of processes. Then, he proposes a new automatic approximation method to tackle it. He uses the proposed method to formally verify that a revised version of the MCS mutual exclusion protocol [12] enjoys desired properties. It is one piece of future work to formally verify with the Farn Wang's method that Anderson protocol enjoys the mutual exclusion property and to compare his method with the technique used in the present paper. It is another piece of future work to formally verify that the MCS mutual exclusion protocol enjoys the mutual exclusion property with the technique used in the present paper.

IX. Conclusion

We summarize some lessons learned from the case study. (1) Abstraction makes it possible to tackle the formal verification task. Although we were not able to formally verify that Anderson protocol enjoys the mutual exclusion property by writing proof scores in CafeOBJ, we were able to conduct the formal verification for A-Anderson protocol, an abstract version of Anderson protocol. (2) Our experience says that once we have written all proof scores to prove that A-Anderson protocol enjoys the property, it is rather straightforward to write the proof scripts for CiMPA. (3) Although CiMPG can automatically generate the proof script for CiMPA from proof scores in CafeOBJ, it takes time to do so. One piece of our future work for (2) is to prepare a gentle guide for non-experts to writing proof scripts for CiMPA from their experiences of writing proof scores in CafeOBJ. Another piece of our future work for (2) and (3) is to come up with better annotations to proof scores for CiMPG to more efficiently generate the proof scripts from annotated proof scores.

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

[3] K. Ogata and K. Futatsugi, "Some tips on writing proof scores in the OTS/CafeOBJ method," in *Algebra, Meaning, and Computation*, 2006, pp. 596–615.

[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] M. Clavel, et al., Ed., *All About Maude*, ser. Lecture Notes in Computer Science. Springer, 2007, vol. 4350.

[7] A. Riesco, K. Ogata, and K. Futatsugi, "A Maude environment for CafeOBJ," *Formal Asp. Comput.*, vol. 29, no. 2, pp. 309–334, 2017.

[8] S. J. Garland and J. V. Guttag, "An overview of LP, the larch power," in *RTA-89*, 1989, pp. 137–151.

[9] K. Ogata and K. Futatsugi, "Simulation-based verification for invariant properties in the OTS/CafeOBJ method," *Electron. Notes Theor. Comput. Sci.*, vol. 201, pp. 127–154, 2008.

[10] K. Ogata and K. Futatsugi, "Specification and verification of some classical mutual exclusion algorithms with CafeOBJ," in *OBJ/CafeOBJ/Maude Workshop at Formal Methods 1999*, 1999, pp. 159–177.

[11] F. Wang, "Automatic verification of pointer data-structure systems for all numbers of processes," in *World Congress on Formal Methods 1999*, 1999, pp. 328–347.

[12] J. M. Mellor-Crummey and M. L. Scott, "Algorithms for scalable synchronization on shared-memory multiprocessors," *ACM Trans. Comput. Syst.*, vol. 9, no. 1, pp. 21–65, 1991.