Master's Thesis

Formal verification of some mutual exclusion protocols with CafeInMaude, its proof assistant and its proof generator

Tran Dinh Duong

Supervisor: Professor Kazuhiro Ogata

Graduate School of Advanced Science and Technology
Japan Advanced Institute of Science and Technology
Information Science

August, 2020

Abstract

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 acquire the permission of using 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. It is important to guarantee that these protocols enjoy the mutual exclusion property and some other desired properties as well. In formal method, theorem proving is one promising technique that can be used to formally verify such problems. This technique especially shows its power when dealing with infinite-state systems, which model checking, although is the most well-known technique in formal method, however, cannot be used.

This thesis uses observational transition systems (OTSs) as state machines and CafeOBJ as a formal specification language to formalize systems (or protocols). Then, we can check whether protocols satisfy some properties by formally verifying that OTSs enjoys such properties. Formal verification, which uses theorem proving as the underlying technique, is essentially done by simultaneous structural induction on a state variable. The verification is conducted in three ways: (1) by writing what are called proof scores and executing them with CafeInMaude, (2) by using CafeInMaude Proof Assistant (CiMPA) to write what are called proof scripts, and (3) by using CafeInMaude Proof Generator (CiMPG) to generate proof scripts from proof scores. CafeInMaude is a tool to introduce CafeOBJ specifications into the Maude system. CiMPA and CiMPG are two extension tools of CafeInMaude. Three ways of verification all have advantages as well as disadvantages. By conducting formal verification in three ways, we triple-check the correctness of our proofs.

Two mutual exclusion protocols: A-Anderson that is an abstract version of Anderson protocol, and MCS are used as two case studies to illustrate the verification techniques. We formally prove that A-Anderson and MCS enjoy the mutual exclusion property. In both case studies, the most intellectual task is lemma conjecture, which is also considered as one of the most challenging problems in theorem proving. This thesis focuses on invariant properties, which are the most basic and important among various kinds of properties. During each invariant proof, we need to conjecture some auxiliary lemmas that are also invariants on the fly. Once we have constructed some good lemmas, the proof can be accomplished straightforwardly; otherwise, it may become unreasonably tough. This thesis also proposes a lemma conjecture

technique that is called Lemma Weakening (LW). The usefulness of LW is demonstrated in the latter case study when conducting formal verification of MCS protocol. Briefly, without the use of LW, we would not have been able to complete the formal proof that MCS enjoys the mutual exclusion property.

Keywords: proof score, algebraic specification language, mutual exclusion protocol, lemma weakening

Acknowledgements

First of all, I would like to express my deep gratitude and appreciation to my supervisor Professor Kazuhiro Ogata for his countless guidance and support. I not only learned from him the knowledge related to my research topic but also always respect him as an ideal model of a researcher that I want to pursue. Without his kindly instruction, it would have been impossible for me to complete this master's program.

I wish to say grateful thanks to Associate Professor Pham Ngoc Hung (VNU University of Engineering and Technology), my former advisor in Vietnam. Although he no longer supervises me directly, we still keep in touch, and I frequently get much valuable advice from him.

I would like to devote my thankful appreciation to all members of Ogata's lab, especially Mr. Do Minh Canh and Mr. Bui Duy Dang for giving me a lot of help in research and daily life since the first day I came to Japan.

With my Vietnamese friends at JAIST, life here becomes less boring. I also received much help from them. Thus, I would like to sincerely thank them, especially Ms. Nguyen Thi Van Anh and Mr. Dang Tran Binh.

I am deeply indebted to a friend in Vietnam, Ms. Le Thi Theu. Without her, maybe I cannot finish this research. Whenever I am depressed because stuck in my work, I usually find her to complain and she always encourages me to continue trying. I cannot say more than thank so much to Theu for her kindly helps.

There is no word that can be used to express my sincere thanks to my family. I would like to thank my parents, my younger sister, and my cousins, especially Mrs. Tran Thi Kim Lien, for all their support to me. I would not make it here without you.

Thank you very much, everyone!
Tran Dinh Duong,
June 17, 2020.

Contents

1 Introduction **1**
- 1.1 Motivation . . . 1
- 1.2 The overview of method . . . 2
- 1.3 Thesis organization . . . 3

2 Preliminaries **5**
- 2.1 Observational Transition Systems (OTSs) . . . 5
- 2.2 CafeOBJ . . . 7
- 2.3 CafeInMaude, CiMPA and CiMPG . . . 9

3 Formal verification of an abstract version of Anderson protocol **11**
- 3.1 Anderson protocol . . . 11
- 3.2 A-Anderson protocol . . . 12
- 3.3 Specification of A-Anderson protocol in CafeOBJ . . . 14
- 3.4 Formal verification with proof scores . . . 17
  - 3.4.1 The mutual exclusion property . . . 17
  - 3.4.2 The other lemmas . . . 22
- 3.5 Formal verification with CiMPA . . . 23
- 3.6 Formal Verification with CiMPG . . . 26

4 Formal verification of MCS protocol **29**
- 4.1 Lemma Strengthening (LS) and Lemma Weakening (LW) . . . 29
- 4.2 MCS list-based queuing lock protocol . . . 31
- 4.3 Specification of MCS protocol in CafeOBJ . . . 32
- 4.4 Formal verification with proof scores . . . 36
  - 4.4.1 Use of Lemma Strengthening (LS) . . . 36
  - 4.4.2 Use of Lemma Weakening (LW) . . . 41
- 4.5 Formal verification with CiMPA . . . 46
- 4.6 Formal verification with CiMPG . . . 48

5 Releated work **50**

6 Conclusion **53**

This thesis was prepared according to the curriculum for the Collaborative Education Program organized by **Japan Advanced Institute of Science and Technology** and **VNU University of Engineering and Technology**.

List of Figures

2.1 The relation between proof score, proof script, CiMPA, and CiMPG 9

3.1 The change of state of Anderson when a process p moves to rs from cs 12
3.2 The change of state of A-Anderson when a process p moves to rs from cs 13
3.3 The proof of `mutex` wrt $\mathcal{S}_{\mathrm{ADS}}$ 18

4.1 The reason why invariant proofs become non-trivial and two approaches to tackling the non-trivial situation 30
4.2 The change of state of MCS when a process p moves to l3 from l2 32
4.3 States υ_{41}, υ_{42}, υ_{43} & υ_{4n} 43

List of Tables

3.1 Case splittings for case (2) in the proof of `mutex` wrt $\mathcal{S}_{\mathrm{ADS}}$ . . 19
3.2 Case splittings for case (3) in the proof of `mutex` wrt $\mathcal{S}_{\mathrm{ADS}}$ . . 20
3.3 Case splittings for case (4) in the proof of `mutex` wrt $\mathcal{S}_{\mathrm{ADS}}$ . . 21
3.4 Sub-goals of `3-9` 24
3.5 Sub-goals of `3-9-1-1-2` 24

4.1 Case splittings for case (2) in the proof of `mutex` wrt $\mathcal{S}_{\mathrm{MCS}}$ . . 37
4.2 Case splittings for case (5) in the proof of `mutex` wrt $\mathcal{S}_{\mathrm{MCS}}$ . . 37
4.3 Case splittings for case (8) in the proof of `mutex` wrt $\mathcal{S}_{\mathrm{MCS}}$ . . 39

Chapter 1

Introduction

1.1 Motivation

Software quality assurance has been considering as an important phase in the software development process life cycle. Ensuring the reliability of a software system is not only a challenging task but also time-, cost-, and effort-consuming. Many approaches could be collaboratively used to guarantee that software systems are truly reliable. One of them is formal verification. This approach uses a state machine to formalize the target system as a mathematical model, and the requirements of that system can be represented as the properties of its formalized state machine. Then, systems verification can be conducted as formal verification of state machine properties. Model checking and theorem proving are two major formal verification techniques, which have been advocating by many researchers. The former can be automatically conducted but basically cannot be used for systems that have an infinite number of states (infinite-state systems) due to state explosion problem. The latter can directly deal with infinite-state systems but it requires human interaction. Conducting theorem proving to formally verify that a system enjoys some desired properties, it often requires us to conjecture some other auxiliary lemmas. This task, which is called lemma conjecture, however, is always considered as one of the most challenging tasks in theorem proving.

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 acquire the permission of using 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. For example, variants of MCS list-based queuing lock protocol (MCS protocol, or simply MCS) [1]

have been used in Java virtual machines. Therefore, guaranteeing that these protocols enjoy some properties, especially, the mutual exclusion property is an important problem; otherwise, it can lead to some serious incidents. For example, on August 14, 2003, a software bug caused by race condition made a widespread power outage throughout parts of the Northeastern and Midwestern United States, and the Canadian province of Ontario [1]. This accident led to a series of other services interrupted such as telephone networks, cellular service, water supply, flights landing, etc. The technique presented in this thesis can be used to check that race condition bug.

1.2 The overview of method

The verification techniques presented in this thesis use theorem proving as the underlying technique. Observational transition systems (OTSs) [2, 3] are used to formalize systems (or protocols) as state machines. CafeOBJ [4], which is an algebraic specification language, is used to specify the OTSs. Formal verifications of invariant properties, which are the most basic and important among various kinds of properties, can be done by writing what are called proof scores in CafeOBJ and executing them with CafeOBJ. Proof scores are essentially developed by simultaneous structural induction on a state variable of the OTS.

CafeInMaude [5] is the second implementation of CafeOBJ in addition to the original implementation, which was done in Common Lisp. CafeInMaude introduces CafeOBJ specifications into the Maude [6] system. It comes with two extension tools CafeInMaude Proof Assistant (CiMPA) and CafeInMaude Proof Generator (CiMPG) [7]. This thesis presents the formal verification in three ways:

(1) by writing proof scores and executing them with CafeInMaude,

(2) by using CafeInMaude Proof Assistant - CiMPA, and

(3) by using CafeInMaude Proof Generator - CiMPG.

CiMPA is a proof assistant that allows users to write what are called proof scripts in order to prove invariant properties on their CafeOBJ specifications. CiMPG provides a minimal set of annotations for identifying proof scores and generating CiMPA scripts for these proof scores. Although the proof score approach (1) is flexible to conduct in a sense of theorem proving, it is also

[1] `https://www.energy.gov/sites/prod/files/oeprod/DocumentsandMedia/BlackoutFinal-Web.pdf`

easy to overlook some cases, leading to incorrect proofs. Using CiMPA (2) to develop the proof by writing proof scripts can get out of this disadvantage. However, it is often the case that CiMPA is not flexible enough to conduct formal verification even though it is not subject to human errors as the proof score approach. CiMPG (3) allows users to combine the flexibility of the proof score approach and the reliability of CiMPA. Given proof scores that should be slightly annotated, CiMPG generates proof scripts for CiMPA. Feeding the generated proof scripts into CiMPA, if CiMPA successfully discharges all goals, the proof scores are correct for the goals. By conducting formal verification in three ways, we triple-check the correctness of our proofs.

There are two mutual exclusion protocols that are used as case studies to demonstrate three verification techniques. The first case study is conducted with an abstract version of Anderson protocol [8], which is called A-Anderson protocol. The second case study presents formal verification with the MCS protocol. We formally verify that both A-Anderson and MCS enjoy the mutual exclusion property.

In both case studies, the most intellectual task is lemma conjecture. This is not surprise since lemma conjecture is always considered as one of the most challenging problems in theorem proving. For each invariant proof, we need to conjecture some lemmas that are also invariants on the fly during the proof. Once some good lemmas are constructed, the proof can be accomplished straightforwardly; otherwise, it may become unreasonably tough. This thesis also proposes a lemma conjecture technique by weakening lemmas. Thus, the technique is called Lemma Weakening (LW). To the best of our knowledge, John Rushby [9] is only the researcher who has used a special form of lemmas weakening called disjunctive invariants. Our way to weaken lemmas is more generic than the Rushby's disjunctive invariants. The usefulness of LW is demonstrated in the second case study when conducting formal verification of MCS protocol. While proving that MCS enjoys the mutual exclusion property, we have realized that LW can make the proof attempt converge that otherwise did not seem to converge in a reasonable amount of time.

1.3 Thesis organization

The remainder of this thesis is organized as follows:

- **Chapter 2 - Preliminaries** gives some common notions and background knowledge that related to formal verification, such as OTSs, invariants, proof score, etc.

- **Chapter 3 - Formal verification of an abstract version of Anderson protocol** presents three ways to formally verify that A-Anderson protocol enjoys the mutual exclusion property.
- **Chapter 4 - Formal verification of MCS protocol** presents the formal verification that MCS protocol enjoys the mutual exclusion property along with the usefulness of LW technique in the proof.
- **Chapter 5 - Related work** mentions some related work.
- **Chapter 6 - Conclusion** summarizes the contribution of this thesis and mentions some future work.

All the specifications, proof scores, proof scripts, etc. presented in this thesis are available at `https://gitlab.com/duongtd23/ms-thesis20`.

Chapter 2

Preliminaries

This chapter gives some common notions and background knowledge which are requirements for the rest of the thesis. We first present the definition of OTSs. Then, we show the basic syntax of CafeOBJ through a simple example. After that, we give descriptions for CafeInMaude, CiMPA, and CiMPG.

2.1 Observational Transition Systems (OTSs)

We suppose that there exists a universal state space denoted Υ and that each data type used in OTSs is provided. The data types include Bool for Boolean values. A data type is denoted D with a subscript such as D_{o1} and D_o.

Definition 2.1. An OTS $\mathcal{S}$ is $\langle \mathcal{O}, \mathcal{I}, \mathcal{T} \rangle$ such that

- $\mathcal{O}$: A finite set of observers. Each *observer* $o : \Upsilon\, D_{o1} \ldots D_{om} \rightarrow D_o$ is a function that takes one state and m (≥ 0) data values and returns one data value. The equivalence relation ($\upsilon_1 =_{\mathcal{S}} \upsilon_2$) between two states $\upsilon_1, \upsilon_2 \in \Upsilon$ is defined as $(\forall o \in \mathcal{O})(\forall x_1 \in D_{o1}) \ldots (\forall x_m \in D_{om})\, (o(\upsilon_1, x_1, \ldots, x_m) = o(\upsilon_2, x_1, \ldots, x_m))$.
- $\mathcal{I}$: The set of initial states such that $\mathcal{I} \subseteq \Upsilon$.
- $\mathcal{T}$: A finite set of transitions. Each *transition* $t : \Upsilon\, D_{t1} \ldots D_{tn} \rightarrow \Upsilon$ is a function that takes one state and n (≥ 0) data values and returns one state, provided that $t(\upsilon_1, y_1, \ldots, y_n) =_{\mathcal{S}} t(\upsilon_2, y_1, \ldots, y_n)$ for each $[\upsilon] \in \Upsilon/=_{\mathcal{S}}$, each $\upsilon_1, \upsilon_2 \in [\upsilon]$ and each $y_i \in D_{ti}$ for $i = 1, \ldots, n$. Each transition t has the condition $c\text{-}t : \Upsilon\, D_{t1} \ldots D_{tn} \rightarrow$ Bool, which is

called *the effective condition* of t. If $c\text{-}t(\upsilon, y_1, \ldots, y_n)$ does not hold, then $t(\upsilon, y_1, \ldots, y_n) =_{\mathcal{S}} \upsilon$.

A pair (υ, υ') of states is called a *transition instance* and υ' is called a *successor state* of υ with respect to (wrt) $\mathcal{S}$ if there exists $t \in \mathcal{T}$ such that $\upsilon' =_{\mathcal{S}} t(\upsilon, y_1, \ldots, y_n)$ with $y_i \in D_{ti}$ where $i \in \{1, \ldots, n\}$. Such a pair (υ, υ') may be denoted $\upsilon \rightarrow_{\mathcal{S}} \upsilon'$ to emphasize that υ directly goes to υ' by one step.

Each state that is reachable from an initial state through transitions is called a reachable state.

Definition 2.2. Given an OTS $\mathcal{S}$, *reachable states* wrt $\mathcal{S}$ are inductively defined:

(i) Each $\upsilon \in \mathcal{I}$ is reachable wrt $\mathcal{S}$.

(ii) For each $t \in \mathcal{T}$ and each $y_k \in D_{tk}$ where $k \in \{1, \ldots, n\}$, $t(\upsilon, y_1, \ldots, y_n)$ is reachable wrt $\mathcal{S}$ if $\upsilon \in \Upsilon$ is reachable wrt $\mathcal{S}$.

$\mathcal{R}_{\mathcal{S}}$ is used to denote the set of all reachable states wrt $\mathcal{S}$.

Predicates whose types are $\Upsilon \rightarrow$ Bool are called *state predicates*. State predicates may have universally quantified variables. State predicates that hold in all reachable states wrt $\mathcal{S}$ are invariants wrt $\mathcal{S}$.

Definition 2.3. A state predicate $\rho : \Upsilon \rightarrow$ Bool is called an *invariant* wrt $\mathcal{S}$ if $\rho(\upsilon)$ is true for all $\upsilon \in \mathcal{R}_{\mathcal{S}}$, i.e. $(\forall \upsilon \in \mathcal{R}_{\mathcal{S}})\, \rho(\upsilon)$.

All properties presented in this thesis are invariants.

Definition 2.4. A state predicate $\rho : \Upsilon \rightarrow$ Bool is called an *inductive invariant* wrt $\mathcal{S}$ if it satisfies the following two conditions:

1. $(\forall \upsilon \in \mathcal{I})\, \rho(\upsilon)$
2. $(\forall t \in \mathcal{T})(\forall \upsilon \in \Upsilon)(\forall y_1 \in D_{t1}) \ldots (\forall y_n \in D_{tn})\, (\rho(\upsilon) \Rightarrow \rho(t(\upsilon, y_1, \ldots, y_n)))$

Informally, we can say that state predicates that are preserved by all transitions are inductive invariants. Inductive invariants wrt $\mathcal{S}$ are invariants wrt $\mathcal{S}$ but not vice versa.

2.2 CafeOBJ

CafeOBJ [4] is an algebraic specification language that can be used to specify various kinds of systems and verify their properties. It provides many advanced features such as the flexible mix-fix syntax, the powerful and clear typing system with ordered sorts (or types), parameterized modules and views for instantiating the parameters. The following `SIMPLE-NAT` module simply defines natural numbers with only the plus operator between two natural numbers in CafeOBJ:

```
mod SIMPLE-NAT {
  [ Zero NzNat < Nat ]
  op 0 : -> Zero {constr}
  op s : Nat -> NzNat {constr}
  op _+_ : Nat Nat -> Nat
  vars M N : Nat
  eq 0 + M = M .
  eq s(M) + N = s(M + N) .
}
```

The module first introduces three sorts (types) `Zero`, `NzNat`, and `Nat` correspond to zero, non-zero numbers, and natural numbers (either zero or non-zero), respectively. The order between them is also indicated such that `Zero` and `NzNat` are sub-sorts of `Nat`. It means that any terms of sort `Zero` or sort `NzNat` also belong to sort `Nat` but not vice versa. `op`, `vars`, and `eq` denote operator, variables, and equation, respectively. In an operator declaration, a list of sort names before "`->`" designates its arity, while the sort name after "`->`" designates its coarity. Operators `0` and `s` are introduced with attribute `constr`, stating that they are constructors of sorts `Zero` and `NzNat`, respectively. This concept plays an important role in theorem proving even though for CafeOBJ, it is just a comment. `0` represents zero, and it is a fresh constant of sort `Zero` since the operator is declared with the empty arity. `s` is the successor function of natural numbers, taking a natural number and returning a non-zero natural number (successor of a natural number n is $n+1$). `0` and `s` are declared in standard operator declarations (e.g, we write `s(0)`), while `_+_` is introduced as an infix operator, thanks to the mid-fix syntax of CafeOBJ. Two underscores in `_+_` represent two natural numbers that are inputs of the plus operator (e.g., we write `0 + s(0)`). The semantic of operators is defined by the last two equations.

To formally verify some desired properties of the systems, users can write what are called proof scores in CafeOBJ and executing them with CafeOBJ. Let us consider again the above example, we can write proof scores to prove

that the plus function is associative, that is `(I + M) + N = I + (M + N)` with arbitrary `I`, `M`, and `N` of sort `Nat`. The proof is done by applying induction on `I`. The following open-close proof fragment corresponds to the base case:

```
open SIMPLE-NAT .
  ops m n : -> Nat .                        -- fresh constants
  red (0 + m) + n = 0 + (m + n) .           -- check base case
close
```

where `--` denotes a comment in CafeOBJ starting from there to the end of that line, `open` makes the given module available, `close` stops the use of the module. The first statement between `open-close` declares two fresh constants of sort `Nat` representing two arbitrary natural numbers. Then, the second statement uses the `red` command to reduce a term standing for the base case. Feeding this open-close fragment into CafeOBJ, CafeOBJ returns `true`, indicating that we have all done with the base case. The open-close proof fragment above is called as the proof score.

Then, the following proof score standing for the induction case:

```
open SIMPLE-NAT .
  ops i m n : -> Nat .                    -- fresh constants
  eq (i + M) + N = i + (M + N) .          -- induction hypothesis
  red (s(i) + m) + n = s(i) + (m + n) . -- check induction case
close
```

The second statement represents the induction hypothesis, that is, with a natural number `i`, the associative property holds. The last statement tries to reduce the induction case, that is, if the associativity holds for `i`, it should also hold for `s(i)` (or `i + 1`). Feeding the proof score into CafeOBJ, CafeOBJ returns `true`. We have completely proved that the plus function of two natuaral numbers is associative.

The proof of associative property, however, is just a very simple example since the proof is directly accomplished by only two open-close fragments. In other non-trivial verification problems including two case studies in this thesis, it often requires us to conduct case splitting or to conjecture some other lemmas to complete the proofs. The former creates a big number of open-close fragments as well as lines of code. The latter, as we mentioned, is always considered as one of the most challenging tasks in theorem proving. We will see more details about these problems in Chapter 3 and Chapter 4.

2.3 CafeInMaude, CiMPA and CiMPG

CafeInMaude [5] is a tool to introduce CafeOBJ [4] specifications into the Maude [6] system. It can be regarded as the second implementation of CafeOBJ in addition to the original one, which was done in Common Lisp. This second implementation has a number of advantages. It improves the performance of some CafeOBJ commands, such as search. It makes CafeOBJ environment easily extensible. That is, it allows us to change the syntax of existing features as well as to introduce new syntax, to add new commands, etc. just by using Maude code, thanks to Maude meta-level features and Full Maude. Using CafeInMaude, we can load the module `SIMPLE-NAT` in Sect. 2.2 into Maude environment. We can also execute the proof scores with CafeInMaude instead of CafeOBJ to prove the associative property of the plus operator.

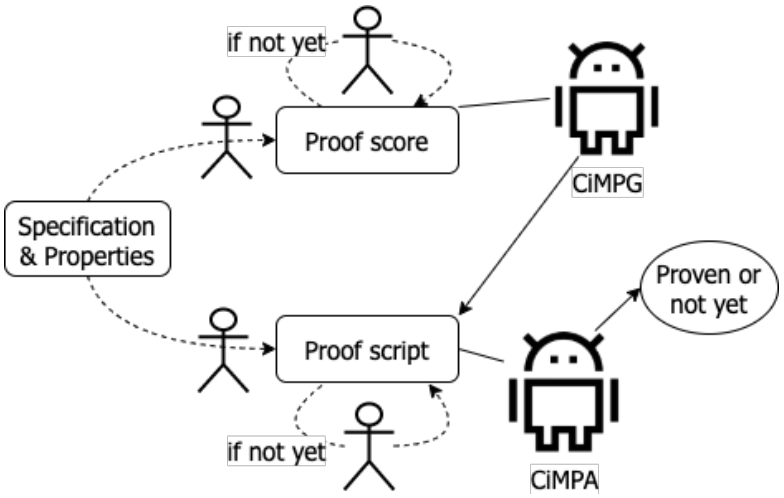


Figure 2.1: The relation between proof score, proof script, CiMPA, and CiMPG

CafeInMaude is introduced with two extension tools: CiMPA and CiMPG. The former is a proof assistant of CafeInMaude that allows users to write what are called proof scripts to prove invariant properties on their CafeOBJ specifications. The latter is a proof generator of CafeInMaude that provides a minimal set of annotations for identifying proof scores and generating CiMPA scripts for these proof scores. Given a specification in CafeOBJ, to check the satisfiability of some desired properties, users can choose to write proof scripts and executing them with CiMPA. If CiMPA finishes successfully, then the properties are proved; otherwise, we need to revise the proof scripts and make the verification attempt again. However, it is often the case that work-

ing with proof scores is easier and more flexible. Therefore, users can decide to write proof scores, then use CiMPG to generate proof scripts for CiMPA. Note that proof scores are subject to human errors such that users can overlook some cases, leading to incorrect proofs. CiMPA can get rid of this kind of flaw. If CiMPA does not work on the proof scripts successfully, the proof scores have something wrong and we need to revise the proof scores, making our verification attempt again. Figure 2.1 graphically describes the relation between the proof score, proof script, CiMPA, and CiMPG.

Chapter 3

Formal verification of an abstract version of Anderson protocol

This chapter presents three ways to formally verify that an abstract version of Anderson protocol enjoys the mutual exclusion property. We first give the details of the Anderson protocol and its abstract version in Sect. 3.1 and Sect. 3.2, respectively. Sect. 3.3 describes how to fomally specify the protocol in CafeOBJ. Then, three last sections present three ways of formal verification.

3.1 Anderson protocol

Anderson protocol [8] 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 fetch&incmod. 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.

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

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

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 thesis. 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.

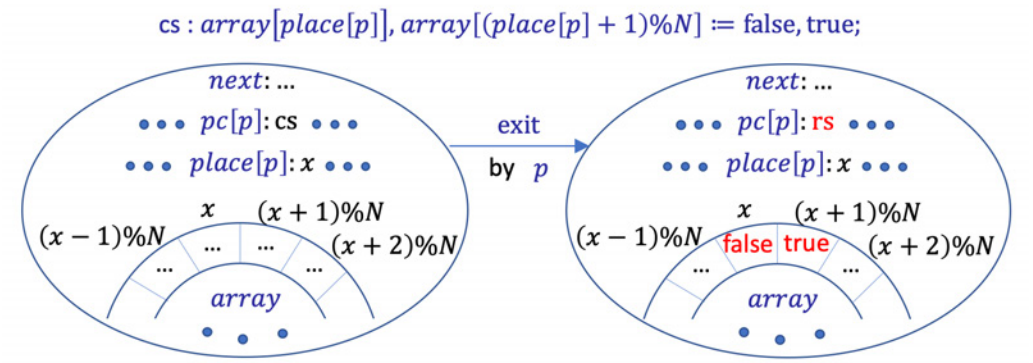


Figure 3.1: The change of state of Anderson when a process p moves to rs from cs

Fig. 3.1 graphically visualizes the change of state when a process p moves to rs from cs. The state, which is represented by the right circle, is the successor state of the one which is represented by the left circle. Note that we use $pc[p]$ to denote the location of process p. The changes are highlighted by red color. Since *array* in Anderson is finite, in the figure, we use a circle (precisely half of a circle) to represent it.

3.2 A-Anderson protocol

It is challenging to formally verify that Anderson protocol enjoys 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. Briefly, these reasons make it so difficult for

us to conjecture some good lemmas to complete the proof. Then, we make an abstract version of the protocol, which is called A-Anderson protocol. In A-Anderson, we use an infinite Boolean array instead of a finite one, fetch&inc instead of fetch&incmod, and no longer use the modulo operation. The assumption that there are N process participating in the protocol is also removed now. It means that the number of process participating in A-Anderson can be infinite.

The pseudo-code of A-Anderson protocol for each process p can be written as follows:

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

where fetch&inc is an atomic operation, taking only one parameter $next$, and 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 thesis, $array[0]$ is true, $array[j]$ is false for any non-zero natural number j and $next$ is 0.

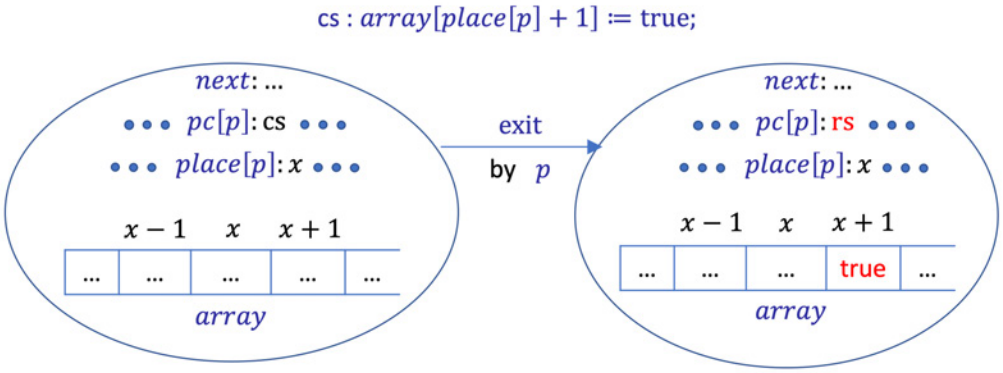


Figure 3.2: The change of state of A-Anderson when a process p moves to rs from cs

Fig. 3.2 graphically visualizes the change of state when a process p moves to rs from cs. The state, which is represented by the right circle, is the successor state of the one which is represented by the left circle. Let us repeat again that we use $pc[p]$ to denote the location of process p and red color is

used to indicate the changes. Since *array* in A-Anderson now becomes an infinite array, we represent it in the straight form with infinity on the right side.

3.3 Specification of A-Anderson protocol in CafeOBJ

In order to specify A-Anderson protocol in CafeOBJ, we first need to define the module `LABEL`. This module is in charge of defining process locations, i.e., rs, ws and cs. The definition starts with `mod!` to indicate that it has tight (initial) semantics. This module defines the sort `Label` and declares three constructors (denoted by the `constr` attribute): `rs`, `ws`, `cs`. It also defines three equations for the equality predicate to indicate that each label is not equal to any others.

```
mod! LABEL {
  [Label]
  ops rs ws cs : -> Label {constr} .
  eq (rs = ws) = false .
  eq (rs = cs) = false .
  eq (ws = cs) = false .
}
```

The module `SIMPLE-NAT` below is in charge of defining natural numbers. In contrast to the module `LABEL`, this module is defined with keyword `mod*`, which indicates that it has loose semantics. It first introduces three sorts `SZero`, `SNzNat` and `SNat` represent zero, non-zero numbers and numbers (either zero or non-zero), respectively. The operation `s` is in charge of defining the successor function of a number, that is it takes a natural number as the input and returns a non-zero number (successor of a natural number n is $n + 1$). The first three equations define some inequalities. For instance, the first equation says that zero is different from any non-zero number. Then, `SIMPLE-NAT` declares a predicate "`<`" that is in charge of defining less-than comparison between two natural numbers. For example, the first equation following the predicate declaration indicates that any natural number is not less than zero.

```
mod* SIMPLE-NAT {
  [ SZero SNzNat < SNat ]
  op 0 : -> SZero {constr}
```

```
  op s : SNat -> SNzNat {constr}
  vars I J L : SNat
  vars K G H : SNzNat
  eq (0 = K) = false .
  eq (0 = s(I)) = false .
  eq (I = s(I)) = false .
  pred _<_ : SNat SNat .
  eq (I < 0) = false .
  eq (0 < K) = true .
  eq (I < s(I)) = true .
  eq (I < s(s(I))) = true .
  eq (I < I) = false .
  ceq (I < s(J)) = true if I < J .
  ceq (s(I) < J) = false if (I < J) = false .
  eq (s(I) = s(J)) = (I = J) .
  ceq (I < s(0)) = false if (I = 0) = false .
  ceq (I < s(J)) = false if (I = J) = false and (I < J) = false .
  ceq (s(I) < s(J)) = true if I < J .
  ceq (I = J) = true if ((I < J) = false and (J < I) = false) .
}
```

Then, the module `ANDERSON` specifies the behavior of the protocol as an OTS $\mathcal{S}_{\mathrm{ADS}}$. It first imports two modules `LABEL`, `SIMPLE-NAT`, and defines the sorts `Sys` and `Pid` representing the set of reachable states and the set of process identifiers, respectively as follows:

```
mod* ANDERSON {
  pr(LABEL + SIMPLE-NAT)
  [Sys] [Pid]
```

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 `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.

A constructor is used to represent 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 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 `I`th 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.

3.4 Formal verification with proof scores

3.4.1 The mutual exclusion property

One desired property A-Anderson protocol should satisfy is the mutual exclusion property. To specify the property, the following module `ADS-INV` is introduced:

```
mod ADS-INV {
  pr(ANDERSON)
  var S : Sys .
  vars P Q : Pid .
  vars G H : SNzNat .
  vars I J K : SNat .
  op mutex : Sys Pid Pid -> Bool
  eq mutex(S,P,Q) =
    ((pc(S,P) = cs and pc(S,Q) = cs) implies (P = Q)) .
}
```

The mutual exclusion property is formalized as the following invariant wrt $\mathcal{S}_{\mathrm{ADS}}$: $(\forall v \in \mathcal{R}_{\mathcal{S}_{\mathrm{ADS}}})(\forall p, q \in \mathtt{Pid})\ \mathtt{mutex}(v, p, q)$. Let us use "the proof of `mutex` wrt $\mathcal{S}_{\mathrm{ADS}}$" (and "to prove `mutex` wrt $\mathcal{S}_{\mathrm{ADS}}$") to mean the proof of $(\forall v \in \mathcal{R}_{\mathcal{S}_{\mathrm{ADS}}})(\forall p, q \in \mathtt{Pid})\ \mathtt{mutex}(v, p, q)$ (and to prove $(\forall v \in \mathcal{R}_{\mathcal{S}_{\mathrm{ADS}}})(\forall p, q \in \mathtt{Pid})\ \mathtt{mutex}(v, p, q)$). This calling is applied for not only $\mathcal{S}_{\mathrm{ADS}}$ or `mutex`, but also other OTSs as well as other similar CafeOBJ operators `inv`i that takes one state variable and zero or more data values and returns a Boolean value in this thesis. We may omit "wrt $\mathcal{S}$" if $\mathcal{S}$ is clear from the context. Note that, initially, `ADS-INV` only has `mutex` that is used to formalized the mutual exclusion property, but while conducting the formal verification, we gradually conjecture lemmas and add them to `ADS-INV` on the fly.

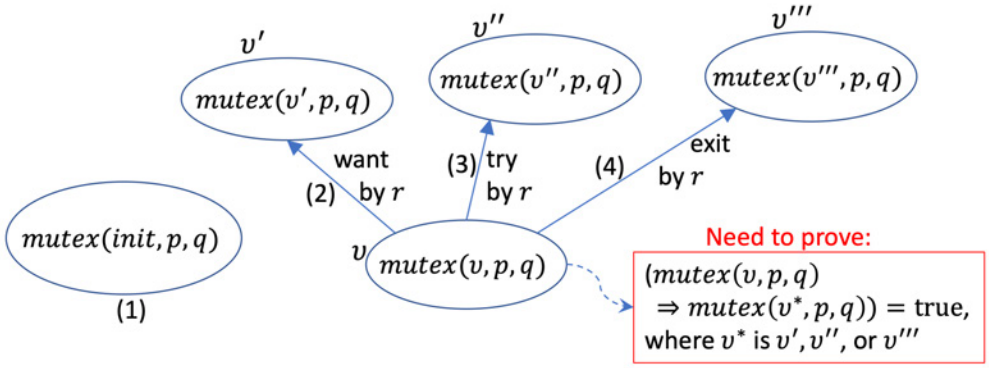


Figure 3.3: The proof of `mutex` wrt $\mathcal{S}_{\mathrm{ADS}}$

The proof of `mutex` wrt $\mathcal{S}_{\mathrm{ADS}}$ is essentially done by applying structural induction on the state variable. The approach is graphically depicted in Fig. 3.3. There are four cases to tackle: base case (1) and three induction cases (2), (3), (4). In (1), we need to show that `mutex` holds in any initial states. In (2), (3) and (4), we want to prove that if `mutex` holds in state v, it

Table 3.1: Case splittings for case (2) in the proof of `mutex` wrt $\mathcal{S}_{\text{ADS}}$

(2.1.1)	`pc(s,r) = rs`, `q = r`
(2.1.2.1)	`pc(s,r) = rs`, `(q = r) = false`, `p = r`
(2.1.2.2)	`pc(s,r) = rs`, `(q = r) = false`, `(p = r) = false`
(2.2)	`(pc(s,r) = rs) = false`

also holds in the successor states υ', υ'' and υ''' of υ, where υ', υ'' and υ''' are made by transitions when an arbitrary process r moves to ws from rs, tries to move to cs from ws and moves to rs from cs, respectively.

Firstly, the proof score of the base case (1) `init` is as follows:

```
open ADS-INV .
  ops p q : -> Pid .
  red mutex(init,p,q) .
close
```

Feeding the proof score into CafeInMaude, CafeInMaude returns `true` meaning that the case is discharged.

Next, let us consider case (2). What to prove is `mutex(want(s,r),p,q)`, where `s` is a fresh constant of `Sys` representing an arbitrary state and `p`, `q` & `r` are fresh constants 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(want(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`. Unlike the base case (1), case (2) can not be discharged directly, instead, it requires us to conduct case splitting. We first split case (2) into two sub-cases: (2.1) `pc(s,r) = rs` and (2.2) `(pc(s,r) = rs) = false`. Case (2.2) can be discharged, its proof score is as follows:

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

Feeding the proof score into CafeInMaude, `true` is returned.

Case (2.1) needs to be split into two sub-cases: (2.1.1) `q = r` and (2.1.2) `(q = r) = false`. Case (2.1.1) can be discharged, while we need to split case (2.1.2) into two sub-cases one more time: (2.1.2.1) `p = r` and (2.1.2.2)

Table 3.2: Case splittings for case (3) in the proof of `mutex` wrt $\mathcal{S}_{\text{ADS}}$

(3.1.1.1)	`pc(s,r) = ws`, `q = r`, `p = r`
(3.1.1.2.1.1)	`pc(s,r) = ws`, `q = r`, `(p = r) = false`, `array(s,place(s,r)) = true`, `pc(s,p) = cs`
(3.1.1.2.1.2)	`pc(s,r) = ws`, `q = r`, `(p = r) = false`, `array(s,place(s,r)) = true`, `(pc(s,p) = cs) = false`
(3.1.1.2.2)	`pc(s,r) = ws`, `q = r`, `(p = r) = false`, `array(s,place(s,r)) = false`
(3.1.2.1.1.1)	`pc(s,r) = ws`, `(q = r) = false`, `p = r`, `array(s,place(s,r)) = true`, `pc(s,q) = cs`
(3.1.2.1.1.2)	`pc(s,r) = ws`, `(q = r) = false`, `p = r`, `array(s,place(s,r)) = true`, `(pc(s,q) = cs) = false`
(3.1.2.1.2)	`pc(s,r) = ws`, `(q = r) = false`, `p = r`, `array(s,place(s,r)) = false`
(3.1.2.2.1)	`pc(s,r) = ws`, `(q = r) = false`, `(p = r) = false`, `p = q`
(3.1.2.2.2.1)	`pc(s,r) = ws`, `(q = r) = false`, `(p = r) = false`, `(p = q) = false`, `array(s,place(s,r)) = true`
(3.1.2.2.2.2)	`pc(s,r) = ws`, `(q = r) = false`, `(p = r) = false`, `(p = q) = false`, `array(s,place(s,r)) = false`
(3.2)	`(pc(s,r) = ws) = false`

`(p = r) = false`. Both sub-cases now can be discharged. The case splittings for case (2) also can be seen through Table 3.1.

So far, we have completely resolved two cases (1) and (2) in the proof of `mutex` wrt $\mathcal{S}_{\text{ADS}}$. These are two simple cases since case (1) is discharged directly and case (2) is discharged just after three times of case spitting without using any other invariants as lemmas. Unfortunately, it is not simple likewise to discharged case (3). In case (3), we need to prove `mutex(try(s,r),p,q)`, where `s` is a fresh constant of `Sys` and `p`, `q` & `r` are fresh constants of `Pid`. Table 3.2 shows the case splittings for case (3) in the proof of `mutex` wrt $\mathcal{S}_{\text{ADS}}$. For example, the open-close fragment of case (3.1.1.2.1.1) is as follows:

```
open ADS-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 mutex(s,p,q) implies mutex(try(s,r),p,q) .
close
```

Feeding the fragment into CafeInMaude, CafeInMaude returns `false`. 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))` is `true`. In that case, process `r` can move to `cs`, breaking the property concerned since 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. `inv1` is declared in the module `ADS-INV` 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) = cs or
  (pc(S,Q) = ws and array(S,place(S,Q)) = true)) = false) .
```

where `S` is a CafeOBJ variable of `Sys`, `P` and `Q` are CafeOBJ variables of `Pid`. The `red` command in the above open-close fragment now becomes as follows:

```
red inv1(s,r,p) implies mutex(s,p,q) implies mutex(try(s,r),p,q) .
```

CafeInMaude now returns `true` for the proof score fragment.

The proof of case (3.1.2.1.1.1) needs the use of `inv1(s,r,q)` as a lemma. The remaining sub-cases of case (3) (in Table 3.2) can be discharged without any lemmas.

Table 3.3: Case splittings for case (4) in the proof of `mutex` wrt $\mathcal{S}_{\text{ADS}}$

(4.1.1)	`pc(s,r) = cs`, `q = r`
(4.1.2.1)	`pc(s,r) = cs`, `(q = r) = false`, `p = r`
(4.1.2.2)	`pc(s,r) = cs`, `(q = r) = false`, `(p = r) = false`
(4.2)	`(pc(s,r) = cs) = false`

Case (4) can be discharged likewise by applying case splittings as shown in Table 3.3. Let us repeat again that `s` is a fresh constant of `Sys`, while `p`, `q` & `r` are fresh contants of `Pid`, and in (4) we need to prove `mutex(exit(s,r),p,q)`. All of the sub-cases can be discharged straightforwardly, without any lemmas.

3.4.2 The other lemmas

Since the proof of `mutex` uses `inv1` as a lemma, we need to prove that `inv1` is also an invariant wrt $\mathcal{S}_{\mathrm{ADS}}$ to complete the formal verification. The proof of `inv1` requires four other invariants that are `inv2`, `inv3`, `mutex` and `inv6`. The proof of `inv3` uses another invariant `inv4` as a lemma. The proof of `inv4` uses another invariant `inv5` as a lemma. `inv2` and `inv5` can be proved independently without the use of any other lemmas. 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. These invariants are declared in the module `ADS-INV` as follows:

```
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) = ws and array(S,place(S,P)) = true)
  or pc(S,P) = cs) 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` in `s(next(S))` and `s(J)` is the successor function of natural numbers. 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. The correctness of this method has been formally proved in the papers [2, 3].

To prove each invariant for an OTS by writing proof scores, 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 CafeInMaude 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 thesis.

3.5 Formal verification with CiMPA

We have presented in the previous section the proof score approach proving that A-Anderson protocol enjoys the mutual exclusion property. The proof score approach is flexible in a sense of theorem proving. The approach, however, has a disadvantage. Proof scores are subjected to human errors. From the previous section, it can be seen that humans can overlook some cases, but they are not pointed out. Using CiMPA can help us get out of this disadvantage.

The proof score approach to formal verification does not require to explicitly construct proof trees. The outcomes of the approach are open-close fragments that correspond to leaf parts of proof trees. Conducing formal verification by writing proof scores, however, we implicitly construct proof trees. Once we have completed formal verification by writing proof scores and executing them with CafeInMaude, we must be able to conduct the formal verification with CiMPA. We partially describe the 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 ADS-INV .
  :goal{
    eq [inv1 :nonexec] : inv1(S:Sys,P:Pid,Q:Pid) = true .
    eq [inv2 :nonexec] : inv2(S:Sys,P:Pid) = true .
    eq [inv3 :nonexec] : inv3(S:Sys,P:Pid,Q:Pid) = true .
    eq [inv4 :nonexec] : inv4(S:Sys,P:Pid) = true .
    eq [inv5 :nonexec] : inv5(S:Sys,P:Pid) = true .
    eq [inv6 :nonexec] : inv6(S:Sys,P:Pid) = true .
    eq [inv7 :nonexec] : inv7(S:Sys) = true .
    eq [inv8 :nonexec] : inv8(S:Sys,I:SNat,J:SNat) = true .
    eq [mutex :nonexec] : mutex(S:Sys,P:Pid,Q:Pid) = true .
  }
```

where `:nonexec` instructs CafeInMaude 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)
```

where `si` stands for simultaneous induction. The command `:apply(si)` starts the proof by applying simultaneous induction on `S`, generating four

Table 3.4: Sub-goals of 3-9

3-9-1-1-1	csb3_9_1, csb3_9_2, csb3_9_3
3-9-1-1-2	csb3_9_1, csb3_9_2, ¬csb3_9_3
3-9-1-2	csb3_9_1, ¬csb3_9_2
3-9-2	¬csb3_9_1

Table 3.5: Sub-goals of 3-9-1-1-2

3-9-1-1-2-1-1	csb3_9_4, csb3_9_5
3-9-1-1-2-1-2	csb3_9_4, ¬csb3_9_5
3-9-1-1-2-2	¬csb3_9_4

sub-goals: 1, 2, 3 and 4 corresponding to exit, init, try and want. Note that, the order between four sub-goals is different from that in the proof score approach since CiMPA generates them in alphabetical order. Each sub-goals consists of nine equations to prove, corresponding to inv1, ..., mutex. 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 as follows:

```
3-1. > TC  eq [inv1 :nonexec]:
  inv1(try(S#Sys,P#Pid),P@Pid,Q@Pid) = true .
...
3-9. TC  eq [mutex :nonexec]:
  mutex(try(S#Sys,P#Pid),P@Pid,Q@Pid) = true .
```

where seven more equations should be written in the place ..., the > symbol indicates that this is the current goal. The command :apply(tc) replaces CafeInMaude variables with fresh constants in goals. S#Sys and P#Pid are fresh constants introduced by :apply(si), while P@Pid and Q@Pid are fresh constants introduced by :apply(tc).

Let us consider goal 3-9, to discharge goal 3-9, we first use the following commands:

```
: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)
```

Based on the three equations, four sub-goals are generated as in Table 3.4. In the table, we use the names of equations (where based on them we apply case splitting) such as `csb3_9_1`, `csb3_9_2` to express for the corresponding equation holds. For example, `csb3_9_1` expresses for `pc(S#Sys,P#Pid) = ws`, and ¬`csb3_9_1` expresses for `(pc(S#Sys,P#Pid) = ws) = false`. For sub-goal `3-9-1-1-1` (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 previous section.

The current goal is now switched to `3-9-1-1-2`. We then use the following commands:

```
: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)
```

Based on one Boolean term and one equation, three sub-goals are generated as in Table 3.5. In the table, we use `csb3_9_4` to express for `array(S#Sys, place(S#Sys,P#Pid)) = true`, and ¬`csb3_9_4` to express for `array(S#Sys, place(S#Sys,P#Pid)) = false`. For sub-goal `3-9-1-1-2-1-1` (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 `P@Pid`, respectively, then the instance is used as a part of 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`, respectively, and the instance is also used as a part of the premise of the implication. After that, `:apply(rd)` is introduced 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 proof score approach presented in Sect. 3.4. The remaining goals can be resolved likewise.

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. The difference is, obviously, we need to use the commands given by CiMPA when conducting formal verification with CiMPA.

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.

3.6 Formal Verification with CiMPG

After writing proof scores to prove 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, 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 more open-close fragment to the proof scores. The open-close fragment is as follows:

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

where `ander` is just an identifier, can be replaced by another one that is more preferred.

Furthermore, we need to add `:id(ander)` in each open-close fragment. For example, the open-close fragment of case (3.1.1.2.1.1) introduced in Sect. 3.4 becomes as follows:

```
open ADS-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 scripts for CiMPA. The generated proof scripts are quite similar to the one written manually. Feeding the generated proof scripts into CiMPA, CiMPA discharges all goals, confirming that the proof scores are correct. In A-Anderson case study, CiMPG took about 626s to generate the proof scripts from the correct proof scores.

This chapter has presented three ways of formal verification that A-Anderson protocol enjoys the mutual exclusion property. Each of the three verification techniques has advantages as well as disadvantages, but by conducting formal verification in three ways, we triple-check the correctness of our proof. We can summarize what has been presented in this chapter as well as some lessons learned from the case study as follows:

- Once we have completely conducted formal verification by writing proof scores, it is rather straightforward to develop the verification with proof scripts and CiMPA.
- Although CiMPG can automatically generate the proof scripts for CiMPA from proof scores, it takes time to do so. However, in the case study, 10 minutes waiting is still acceptable, at least in comparison with writing proof scripts manually.
- Abstraction makes it possible to achieve formal verification. Three verification techniques helped us complete the formal verification of A-Anderson, but could not achieve the same result with Anderson. However, we have successfully used "simulation-based verification for invariant properties [10]" so as to formally verify that Anderson enjoys the mutual exclusion property even though this part is not presented in this thesis.

- Auxiliary lemmas are required in all three ways of formal verification. Lemma conjecture is the most intellectual task while conducting formal verification of A-Anderson. In the upcoming chapter, a lemma conjecture technique will be introduced.

Chapter 4

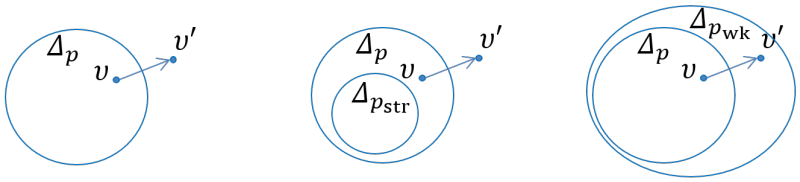
Formal verification of MCS protocol

This chapter describes how to formally verify that MCS protocol enjoys the mutual exclusion property. We first present a lemma conjecture technique called Lemma Weakening (LW). Then, the usefulness of this technique is demonstrated along with the formal verification of MCS.

4.1 Lemma Strengthening (LS) and Lemma Weakening (LW)

Suppose that we want to prove that a state predicate p is an invariant wrt an OTS $\mathcal{S}$. It is often the case that p is not inductive and then there is a transition instance that does not preserve p such that $p(o)$ holds but $p(o')$ does not, where $o \rightarrow o'$ is a transition instance, which is shown in Fig. 4.1 (a), where Δ_p is $\{o \in \Upsilon \mid p(o)\}$. This is the reason why invariant proofs become non-trivial or even can become very hard. If we can successfully show that the source o is not reachable wrt $\mathcal{S}$, then we do not need to consider the transition instance, being able to discharge the case. One possible way to do so is to find p_{str} that is stronger than p such that $p_{\mathrm{str}}(o)$ does not hold and to prove that p_{str} is an invariant wrt $\mathcal{S}$, which is shown in Fig. 4.1 (b). If p_{str} is inductive wrt $\mathcal{S}$, we do not need to use any more lemmas. This approach has been summarized as the proof rule Inv by Manna and Pnueli [11].

To prove that p is an invariant of $\mathcal{S}$ (or a state machine), in general, we need to find an inductive invariant q wrt $\mathcal{S}$ such that $q(o) \Rightarrow p(o)$ for all states $o \in \Upsilon$. In practice, q is often in the form $p \wedge p'$, and p' is often in the form $q_1 \wedge \ldots \wedge q_n$. $q_1, \ldots, q_n$ are the lemmas of the proof that p is an invariant wrt $\mathcal{S}$. It is often the case that we do not know any of $q_1, \ldots, q_n$



(a) The source making the proof non-trivial

(b) p_{str} obtained by strengthening p

(c) p_{wk} obtained by weakening p

Figure 4.1: The reason why invariant proofs become non-trivial and two approaches to tackling the non-trivial situation

in advance. We need to gradually conjecture $q_1, \ldots, q_n$ one by one when we encounter the situation shown in Fig. 4.1 (a). For example, in the previous Chapter, to prove that `mutex` is an invariant wrt $\mathcal{S}_{\mathrm{ADS}}$, it requires to use `inv1` as a lemma, and during the proof of `inv1`, we conjecture some more other lemmas. `mutex` corresponds to p, while `mutex` $\wedge$ `inv1` $\wedge \ldots \wedge$ `inv8` corresponds to the inductive invariant q. $q_i(v)$ for the source v needs not to hold but does not need to be properly stronger than p because $p \wedge q_1 \wedge \ldots \wedge q_n$ is surely stronger than p. In general, when we have conjectured up to q_k, we do not know how many more lemmas we need to conjecture.

The reason why invariant proofs become non-trivial or even can become very hard is because there exists a transition instance $v \to v'$ as shown in Fig. 4.1 (a). The proof rule Inv gets rid of such a transition instance as shown in Fig. 4.1 (b). Another possible way to get rid of such a transition instance is to find p_{wk} that is weaker than p such that $p_{\mathrm{wk}}(v')$ holds and to prove that p_{wk} is an invariant wrt $\mathcal{S}$, which is shown in Fig. 4.1 (c). Even though p_{wk} is an invariant wrt $\mathcal{S}$, however, it does not guarantee that p is an invariant wrt $\mathcal{S}$. This is because Δ_p may not contain all reachable states in $\mathcal{R}_{\mathcal{S}}$. Therefore, the second approach is not used to prove that p is an invariant wrt $\mathcal{S}$. This might be the reason why the second approach has been rarely used. Although the second approach is not very useful for p, it may be useful for some q_i, a lemma of the proof that p is an invariant wrt $\mathcal{S}$. In this thesis, strengthening lemmas q_i is called Lemma Strengthening (LS), while weakening lemmas q_i is called Lemma Weakening (LW).

While proving that MCS enjoys the mutual exclusion property, we have encountered a situation where use of only LS did not seem to make the proof converge. We got over the situation by using LW to weaken some lemmas. Note that we have used both LS and LW for the proof that MCS enjoys the mutual exclusion property. We will describe in which way LW makes the

proof attempt converge in Sect. 4.4. Two upcoming sections describe MCS and formal specification of $\mathcal{S}_{\text{MCS}}$ that formalizes the protocol.

4.2 MCS list-based queuing lock protocol

MCS is a mutual exclusion protocol invented by Mellor-Crummey and Scott [1]. Variants of MCS have been used in Java VMs and therefore the 2006 Edsger W. Dijkstra Prize in Distributed Computing went to their paper [1]. This protocol has the following properties:

- it guarantees FIFO ordering of lock acquisitions;
- it spins on locally-accessible flag variable only;
- it requires a small constant amount of space per lock; and
- it works equally well (requiring only $O(1)$ network transactions per lock acquisiion) on machines with and without coherent caches.

The pseudo-code of MCS protocol for each process p can be written as follows:

rs : "Remainder Section"
l1 : $next_p$:= nop;
l2 : $prede_p$:= fetch&store($glock, p$);
l3 : **if** $prede_p \neq$ nop {
l4 : $lock_p$:= true;
l5 : $next_{prede_p}$:= p;
l6 : **repeat while** $lock_p$; }
cs : "Critical Section"
l7 : **if** $next_p$ = nop {
l8 : **if** comp&swap($glock, p$, nop)
l9 : **goto** rs;
l10 : **repeat while** $next_p$ = nop; }
l11 : $lock_{next_p}$:= false;
l12 : **goto** rs;

There is one global variable *glock* shared by all processes participating in MCS protocol. Its value is a process ID (Pid) or nop, a dummy process ID. *glock* basically refers to the bottom element if the queue is not empty. Each process p maintains three local variables $next_p$, $lock_p$ and $prede_p$ whose types are Pid, Bool and Pid, respectively. Initially, *glock*, $next_p$, $lock_p$ and $prede_p$ are nop, nop, false and nop, respectively. $next_p$ and $prede_p$ are used to construct a global queue of processes (or process IDs). Basically, $next_p$

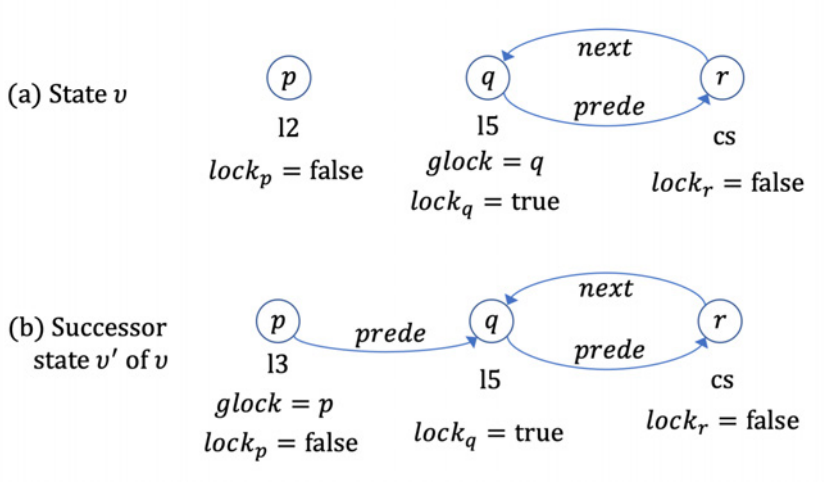


Figure 4.2: The change of state of MCS when a process p moves to l3 from l2

refers to the next element of the queue if p is in the queue. Note that $next_p$ may be nop even though p is not the bottom element of the queue. $prede_p$ refers to the predecessor element of the queue. $lock_p$ is the local lock on which process p is spinning while $lock_p$ is true to wait for entering the critical section.

MCS uses two non-trivial atomic instructions: fetch&store and comp&swap. For a variable x and a value α, fetch&store(x, α) atomically does the following: x is set to α and the old value of x is returned. For a variable x and values α, β, comp&swap(x, α, β) atomically does the following: if x equals α, then x is set to β and true is returned; otherwise false is just returned.

Fig. 4.2 graphically visualizes the change of state of MCS when a process p moves to l3 from l2. In the state υ, which is represented by Fig. 4.2 (a), processes p, q, and r, located at l2, l5, and cs, respectively; $glock$ is q; $next$ of r is q; and $prede$ of q is r. When process p moves to $l3$, $glock$ is set to itself, and its $prede$ is set to q (depicted in Fig. 4.2 (b)).

4.3 Specification of MCS protocol in CafeOBJ

In order to specify MCS protocol in CafeOBJ, we first need to define the module `LABEL`, which is in charge of defining process locations. This module first introduces the sort `Label` and declares fourteen constructors (denoted by the `constr` attribute): `rs`, `l1`, `l2`, `l3`, `l4`, `l5`, `l6`, `cs`, `l7`, `l8`, `l9`, `l10`, `l11`

and l12. It also defines many equations for the equality predicate to indicate that each label is not equal to any others.

```
mod! LABEL {
  [Label]
  ops l1 l2 l3 l4 l5 l6 l7 l8 l9 l10 l11 l12 rs cs : -> Label
    {constr} .
  eq (l1 = l2) = false . eq (l1 = l3) = false .
  ...
  eq (l1 = rs) = false . eq (l1 = cs) = false .
  eq (l2 = l3) = false . eq (l2 = l4) = false .
  ...
  eq (l2 = cs) = false . eq (l3 = l4) = false .
  ...
  eq (l12 = cs) = false . eq (rs = cs) = false .
}
```

where many other equations should be written in the place ..., constr indicates that rs, l1, l2,..., l12 are constructors of sort Label.

The module PID is in charge of defining process identifiers. Sort Pid represents the (correct) process identifiers, while sort Nop expresses for the erroneous identifiers which are the dummy processes. Sort Pid&Nop is a supersort of sorts Pid and Nop. The module also states that any correct identifier is different from nop.

```
mod* PID {
  [Nop Pid < Pid&Nop]
  op nop : -> Nop {constr} .
  var P : Pid .
  eq (P = nop) = false .
}
```

Finally, the module MCS specifies the behavior of the protocol as an OTS $\mathcal{S}_{\mathrm{MCS}}$. It first imports two modules LABEL, PID, and defines the sort Sys representing the set of reachable states as follows:

```
mod* MCS {
  pr(LABEL + PID)
  [Sys]
```

Each state of MCS protocol can be characterized by the following pieces of information: the value of *glock*; the location, next process, previous process, lock value of each process. Therefore, the following observation functions are used:

```
op glock : Sys -> Pid&Nop .
op pc : Sys Pid -> Label .
op next : Sys Pid -> Pid&Nop .
op prede : Sys Pid -> Pid&Nop .
op lock : Sys Pid -> Bool .
```

since the values of `glock`, `next_p`, `prede_p` can be nop, thus we need to use sort `Pid&Nop` instead of sort `Pid`. Then, a constructor is used to represent an arbitrary initial state as follows:

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

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

```
eq glock(init) = nop .
eq pc(init,P) = rs .
eq next(init,P) = nop .
eq prede(init,P) = nop .
eq lock(init,P) = false .
```

where `P` is a CafeOBJ variable of `Pid`. Fourteen following functions specify the transitions of MCS:

```
-- moves to l1 from rs
op want  : Sys Pid -> Sys {constr} .
-- moves to l2 from l1
op stnxt : Sys Pid -> Sys {constr} .
-- moves to l3 from l2
op stprd : Sys Pid -> Sys {constr} .
-- moves to l4 or cs from l3
op chprd : Sys Pid -> Sys {constr} .
-- moves to l5 from l4
op stlck : Sys Pid -> Sys {constr} .
-- moves to l6 from l5
op stnpr : Sys Pid -> Sys {constr} .
-- tries to move to cs from l6
op chlck : Sys Pid -> Sys {constr} .
-- moves to l7 from cs
op exit  : Sys Pid -> Sys {constr} .
-- moves to l8 or l11 from l7
op chnxt : Sys Pid -> Sys {constr} .
-- moves to l9 or l10 from l8
op chglk : Sys Pid -> Sys {constr} .
```

```
-- moves to rs from l9
op go2rs  : Sys Pid -> Sys {constr} .
-- tries to move to l11 from l10
op chnxt2 : Sys Pid -> Sys {constr} .
-- moves to l12 from l11
op stlnx  : Sys Pid -> Sys {constr} .
-- moves to rs from l12
op go2rs2 : Sys Pid -> Sys {constr} .
```

Each of the above fourteen functions is followed by a comment line that describes the corresponding transition of a process it captured. For example, if a process p is located at l1 in state s, then `stnxt`(s, p) denotes the state just after p has executed the statement at l1 and moves to l2 from l1. The reachable states are composed of these fourteen transition functions plus `init` function.

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

`want` transition is defined as follows:

```
eq glock(want(S,P)) = glock(S) .
ceq pc(want(S,P),Q) = (if P = Q then l1 else pc(S,Q) fi)
  if pc(S,P) = rs .
eq next(want(S,P),Q) = next(S,Q) .
eq lock(want(S,P),Q) = lock(S,Q) .
eq prede(want(S,P),Q) = prede(S,Q) .
ceq want(S,P) = S if (pc(S,P) = rs) = false .
```

the equations say that if the location of process `P` currently is `rs`, then `P`'s location changes to `l1`, the location of each other process `Q` does not change; `next`, `lock`, `prede` of every processes and `glock` do not change. If currently process `P` is not located at `rs`, nothing changes.

In the same way, `chlck` transition is defined as follows:

```
eq glock(chlck(S,P)) = glock(S) .
ceq pc(chlck(S,P),Q)
  = (if P = Q then
       (if lock(S,P) then l6 else cs fi)
     else
       pc(S,Q) fi)
  if pc(S,P) = l6 .
eq next(chlck(S,P),Q) = next(S,Q) .
eq lock(chlck(S,P),Q) = lock(S,Q) .
eq prede(chlck(S,P),Q) = prede(S,Q) .
ceq chlck(S,P) = S if (pc(S,P) = l6) = false .
```

the equations say that if currently the location of process P is l6 and lock of P is true, then P's location changes to cs. next, lock, prede of every processes and glock do not change. If currently process P is not located at l6, nothing changes.

The remaining transitions can be defined likewise. All of them can be found at the webpage presented in Chapter 1.

4.4 Formal verification with proof scores

This section presents the proof score approach to formal verification that MCS protocol enjoys the mutual exclusion property. Since the complete proof is quite long, we will only present here a part of it. We will separately describe the use of LS and LW in the verification.

4.4.1 Use of Lemma Strengthening (LS)

Similar to the A-Anderson case study, we introduce the module MCS-INV to specify the mutual exclusion property. The module is as follows:

```
mod MCS-INV {
  pr(MCS)
  var S : Sys
  vars P Q : Pid
  op mutex : Sys Pid Pid -> Bool
  eq mutex(S,P,Q) =
    ((pc(S,P) = cs and pc(S,Q) = cs) implies (P = Q)) .
}
```

The equation mutex says that there is always at most one process located at the critical section at any given time.

mutex(S,P,Q) is proved for all reachable states S and all process IDs P & Q by structural induction on S. There are fifteen cases to tackle including: (1) init, (2) want, (3) stnxt, (4) stprd, (5) chprd, (6) stlck, (7) stnpr, (8) chlck, (9) exit, (10) chnxt, (11) chglk, (12) go2rs, (13) chnxt2, (14) stlnx, (15) go2rs2. Similar to the A-Anderson case study, the base case (1) init can be discharged straightforwardly. Considering case (2), we need to prove mutex(want(s,r),p,q), where s is a fresh constant of Sys representing an arbitrary state and p, q and r are fresh constants of Pid representing arbitrary process IDs. The induction hypothesis is mutex(s,P,Q) for all process IDs P & Q. Let us repeat again that s is shared by mutex(want(s,r),p,q) and mutex(s,P,Q), while the variables P and Q can be replaced with any

Table 4.1: Case splittings for case (2) in the proof of `mutex` wrt $\mathcal{S}_{\text{MCS}}$

(2.1.1)	`pc(s,r) = rs`, `q = r`
(2.1.2.1)	`pc(s,r) = rs`, `(q = r) = false`, `p = r`
(2.1.2.2)	`pc(s,r) = rs`, `(q = r) = false`, `(p = r) = false`
(2.2)	`(pc(s,r) = rs) = false`

terms of `Pid`, such as `p` and `q`. Case (2) is discharged by splitting it into four sub-cases as shown in Table 4.1.

It is not always straightforward likewise to discharged the thirteen remaining cases. We are required either to split cases much more times or to conjecture new lemma to discharge a case. The former creates a huge number of open-close fragments as well as lines of code. The latter is always considered as one of the most challenging tasks in theorem proving. Next, let us consider case (5) as the evidence for this argument.

Table 4.2: Case splittings for case (5) in the proof of `mutex` wrt $\mathcal{S}_{\text{MCS}}$

(5.1.1.1)	`pc(s,r) = l3`, `p = r`, `q = r`
(5.1.1.2.1.1)	`pc(s,r) = l3`, `p = r`, `(q = r) = false`, `prede(s,r) = nop`, `pc(s,q) = cs`
(5.1.1.2.1.2)	`pc(s,r) = l3`, `p = r`, `(q = r) = false`, `prede(s,r) = nop`, `(pc(s,q) = cs) = false`
(5.1.1.2.2)	`pc(s,r) = l3`, `p = r`, `(q = r) = false`, `(prede(s,r) = nop) = false`
(5.1.2.1.1.1)	`pc(s,r) = l3`, `(p = r) = false`, `q = r`, `prede(s,r) = nop`, `pc(s,p) = cs`
(5.1.2.1.1.2)	`pc(s,r) = l3`, `(p = r) = false`, `q = r`, `prede(s,r) = nop`, `(pc(s,p) = cs) = false`
(5.1.2.1.2)	`pc(s,r) = l3`, `(p = r) = false`, `q = r`, `(prede(s,r) = nop) = false`
(5.1.2.2)	`pc(s,r) = l3`, `(p = r) = false`, `(q = r) = false`
(5.2)	`(pc(s,r) = l3) = false`

In case (5), what we need to prove is `mutex(chprd(s,r),p,q)`, where `s` is a fresh constant of `Sys` and `p`, `q` & `r` are fresh contants of `Pid`. Table 4.2 shows

the case splittings for case (5). Let us consider sub-case (5.1.1.2.1.1). In (5.1.1.2.1.1), `mutex(s,p,q)` reduces to `true`, while `mutex(chprd(s,r),p,q)` reduces to `false`, and then `mutex(s,p,q) implies mutex(chprd(s,r),p,q)` reduces to `false`. The pair of states `s` and `chprd(s,r)` is a transition instance as shown in Fig. 4.1 (a). We need to conjecture and use a lemma to discharge (5.1.1.2.1.1).

One possible lemma conjectured most straightforwardly can be constructed by combining the equations that characterize (5.1.1.2.1.1) with conjunction, negating the whole formula and replacing the fresh constants `s`, `p`, `q` & `r` with variables `S`, `P`, `Q` & `R` [12]. Since the formula constructed is in the form `(not P = R) or` F that is equivalent to `(P = R) implies` F. Thus, the lemma constructed is F in which R is replaced with P, which is equivalent to the following:

```
eq inv1'(S,P,Q) = ((pc(S,P) = l3 and prede(S,P) = nop and
  (P = Q) = false) implies (pc(S,Q) = cs) = false) .
```

Since `inv1'(s,p,q)` reduces to `false`, we can use `inv1'` as a lemma to discharge (5.1.1.2.1.1) as follows:

```
red inv1'(s,p,q) implies mutex(s,p,q) implies mutex(chprd(s,r),p,q) .
```

In the proof of `inv1'`, however, we encounter a sub-case (1'.8.1.1.2.2.1.1) in which CafeInMaude returns `false` for its proof score. The open-close fragment is as follows:

```
open MCS-INV .
  op s : -> Sys .
  ops p q r : -> Pid .
  eq pc(s,r) = l6 .
  eq p = r .
  eq (q = r) = false .
  eq lock(s,r) = false .
  eq pc(s,q) = l3 .
  eq prede(s,q) = nop .
  red inv1'(s,p,q) implies inv1'(chlck(s,r),p,q) .
close
```

By applying the same technique that has been explained above to conjecture `inv1'`, we can conjecture another lemma to discharge (1'.8.1.1.2.2.1.1) as follows:

```
eq inv1''(S,P,Q) = ((pc(S,Q) = l3 and prede(S,Q) = nop and (P = Q)
  = false) implies (pc(S,P) = l6 and lock(S,P) = false) = false) .
```

The conditional part of `inv1''` is exactly the same as that of `inv1'`. The reason why we use the forms of `inv1'` and `inv1''` is because we emphasize what are shared by `inv1'` and `inv1''`. The proof of `inv1''` needs yet another lemma whose conditional part is exactly the same as that of `inv1'`. We need several different lemmas when we only use the most straightforward lemmas. Therefore, we strengthen them to obtain the following lemma:

```
eq inv1(S,P,Q) = ((pc(S,Q) = l3 and prede(S,Q) = nop and
  (P = Q) = false) implies ((pc(S,P) = cs or pc(S,P) = l7 or
  pc(S,P) = l8 or pc(S,P) = l10 or pc(S,P) = l11 or
  (pc(S,P) = l6 and lock(S,P) = false)) = false)) .
```

And the `red` command in the open-close fragment of case (5.1.1.2.1.1) now becomes as follows:

```
red inv1(s,p,q) implies mutex(s,p,q) implies mutex(chprd(s,r),p,q) .
```

The proof of `inv1` needs totally less lemmas than that of `inv1'`. More precisely, the set of lemmas that need to be used in the proof of `inv1` is a subset of those in the proof of `inv1'`.

Table 4.3: Case splittings for case (8) in the proof of `mutex` wrt $\mathcal{S}_{\text{MCS}}$

(8.1.1.1)	`pc(s,r) = l6`, `p = r`, `q = r`
(8.1.1.2.1)	`pc(s,r) = l6`, `p = r`, `(q = r) = false`, `lock(s,r) = true`
(8.1.1.2.2.1)	`pc(s,r) = l6`, `p = r`, `(q = r) = false`, `lock(s,r) = false`, `pc(s,q) = cs`
(8.1.1.2.2.2)	`pc(s,r) = l6`, `p = r`, `(q = r) = false`, `lock(s,r) = false`, `(pc(s,q) = cs) = false`
(8.1.2.1.1)	`pc(s,r) = l6`, `(p = r) = false`, `q = r`, `lock(s,r) = true`
(8.1.2.1.2.1)	`pc(s,r) = l6`, `(p = r) = false`, `q = r`, `lock(s,r) = false`, `pc(s,q) = cs`
(8.1.2.1.2.2)	`pc(s,r) = l6`, `(p = r) = false`, `q = r`, `lock(s,r) = false`, `(pc(s,q) = cs) = false`
(8.1.2.2)	`pc(s,r) = l6`, `(p = r) = false`, `(q = r) = false`
(8.2)	`(pc(s,r) = l6) = false`

The proof of case (8) also requires another lemma. Table 4.3 shows the case splittings for case (8). Let us repeat again that `s` is a fresh constant

of Sys, while p, q & r are fresh contants of Pid, and in case (8) we need to prove mutex(chlck(s,r),p,q). Sub-case (8.1.1.2.2.1) uses inv6(s,q,p) as a lemma, and sub-case (8.1.2.1.2.1) uses inv6(s,p,q) as a lemma. inv6 is declared in the module MCS-INV as follows:

```
eq inv6(S,P,Q) = ((pc(S,Q) = l6 and lock(S,Q) = false and
  (P = Q) = false) implies ((pc(S,P) = cs or pc(S,P) = l7 or
  pc(S,P) = l8 or pc(S,P) = l10 or pc(S,P) = l11 or
  (pc(S,P) = l6 and lock(S,P)= false)) = false)) .
```

Since inv1 and inv6 are used in the proof of mutex wrt $\mathcal{S}_{\mathrm{MCS}}$, they need to be proved that they are also invariants wrt $\mathcal{S}_{\mathrm{MCS}}$ to complete the formal verification. The proof of inv1 uses inv2, inv3 and inv4 as lemmas. The proof of inv2 uses inv4 as a lemma. The proof of inv3 uses only inv5 as a lemma and vice versa. The proof of inv4 and inv6 use inv1, inv3 and inv7 as lemmas. The proof of inv7 uses inv1, inv3 and inv6 as lemmas. The remaining invariants are declared in module MCS-INV as follows:

```
eq inv2(S,P,Q) = ((pc(S,P) = l3 and prede(S,P) = nop and pc(S,Q) = l3
  and (P = Q) = false) implies (prede(S,Q) = nop) = false) .

eq inv3(S,P) = (pc(S,P) = l5 implies lock(S,P) = true) .

eq inv4(S,P) = ((pc(S,P) = l11 or pc(S,P) = l10 or pc(S,P) = l8 or
  pc(S,P) = l7 or (pc(S,P) = l6 and lock(S,P) = false) or
  (pc(S,P) = l3 and prede(S,P) = nop) or pc(S,P) = cs) implies
  (glock(S) = nop) = false) .

eq inv5(S,P,Q) = ((next(S,Q) = P and (P = Q) = false and
  (pc(S,Q) = l12 or pc(S,Q) = l1 or pc(S,Q) = rs) = false) implies
  (pc(S,P) = l6 and lock(S,P) = true and prede(S,P) = Q)) .

eq inv7(S,P,Q) = (((pc(S,Q) = l11 or pc(S,Q) = l10 or pc(S,Q) = l8
  or pc(S,Q) = l7 or pc(S,Q) = cs) and (P = Q) = false) implies
  ((pc(S,P) = cs or pc(S,P) = l7 or pc(S,P) = l8 or pc(S,P) = l10 or
  pc(S,P) = l11 or (pc(S,P) = l6 and lock(S,P) = false)) = false)) .
```

Let us partially give the explanation for inv4. inv4 says that if there exists a process P located at cs, or l7, or l8, or l10, or l11, or l6 with lock of P is false, or l3 with prede of P is nop; then glock can not be nop. The remaining lemmas can be understood likewise. Let us repeat again that we did not come up with the seven lemmas from the beginning, but we have gradually constructed each of them while conducting formal verification. For example, in the proof of mutex, inv1 is constructed; or inv2 together with inv3 and inv4 are constructed when we try to prove inv1.

4.4.2 Use of Lemma Weakening (LW)

While proving that MCS enjoys the mutual exclusion property, use of LS (but not use of LW), together with case splitting, etc., did not seem to make our proof attempt converge. Use of LW made it converge. We illustrate the use of LW in the two cases in details.

Case 1

Initially, we used the following `inv40` instead of `inv4`:

```
eq inv40(S,P) = ((pc(S,P) = l3 or pc(S,P) = l4 or pc(S,P) = l5 or
  pc(S,P) = l6 or pc(S,P) = cs or pc(S,P) = l7 or pc(S,P) = l8 or
  pc(S,P) = l10 or pc(S,P) = l11) implies (glock(S) = nop) = false) .
```

`inv40` is obtained by strengthening `inv4`. Let us note that `40` in the notation `inv40` does not mean that there are 40 or more invariants that have been conjectured. We put confidence in that whenever there exists a process `P` located at `l3` or `l6` (or `cs`, or `l7`, or `l8`, or `l10`, or `l11` as well), `glock` can not be `nop`. Thus, we strongly believe that strengthening `inv4` to obtain `inv40` is the correct way to complete the formal verification (similar to the way we strengthen `inv1'` and `inv1''` to obtain `inv1`). Accordingly, we believe that `inv40` is truly an invariant wrt $\mathcal{S}_{\mathrm{MCS}}$. Let us consider a sub-case of the induction case `chglk` for the proof attempt of `inv40`. The open-close fragment of the sub-case is as follows:

```
open MCS-INV .
  op s : -> Sys . ops p r : -> Pid .
  eq pc(s,r) = l8 . eq (p = r) = false .
  eq glock(s) = r . eq pc(s,p) = l3 .
  red inv40(s,p) implies inv40(chglk(s,r),p) .
close
```

Let υ_{40} be an arbitrary state in which the four equations used in the fragment hold. CafeInMaude returns `false` for the fragment. This is why we need a lemma to discharge the sub-case. By strengthening the lemma constructed straightforwardly from the four equations used in the fragment, we obtain the following lemma:

```
eq inv41(S,P,Q) = ((pc(S,P) = l3 or pc(S,P) = l4 or pc(S,P) = l5 or
  pc(S,P) = l6 or pc(S,P) = cs or pc(S,P) = l7 or pc(S,P) = l8 or
  pc(S,P) = l10 or pc(S,P) = l11) and  glock(S) = Q and
  (P = Q) = false) implies (pc(S,Q) = cs or pc(S,Q) = l7 or
  pc(S,Q) = l8 or pc(S,Q) = l10 or pc(S,Q) = l11 or
  (pc(S,Q) = l6 and lock(S,Q) = false)) = false .
```

inv41 can be used as a lemma to discharge the sub-case. During the proof of inv41, we encounter three sub-cases of the induction case stlnx. One of the three sub-cases has the open-close fragment as follows:

```
open MCS-INV .
  eq pc(s,r) = l11 .
  eq next(s,r) = q .
  eq glock(s) = q .
  eq pc(s,p) = l3 . eq pc(s,q) = l6
  eq lock(s,q) = true .
  eq (p = r) = false . eq (q = r) = false . eq (p = q) = false .
  red inv41(s,p,q) implies inv41(stlnx(s,r),p,q) .
close
```

The nine equations characterize the sub-case. The other two sub-cases are characterized by almost the same equations. The only difference is pc(s,p) = l3, instead of which pc(s,p) = l4 and pc(s,p) = l5 hold for the other two sub-cases, respectively. Let υ_{41} be an arbitrary state that corresponds to any of the three sub-cases. The proofs of each three sub-cases reduces to false, and then we need a lemma to discharge the three sub-cases. By applying the same technique that has been explained above to construct inv41, we obtain the following lemma:

```
eq inv42(S,P,Q,R) = ((pc(S,R) = cs or pc(S,R) = l7 or pc(S,R) = l8
  or pc(S,R) = l10 or pc(S,R) = l11 or (pc(S,R) = l6 and lock(S,R)
  = false)) and glock(S) = Q and next(S,R) = Q and (P = R) = false
  and (Q = R) = false and (P = Q) = false) implies (pc(S,P) = l3 or
  pc(S,P) = l4 or pc(S,P) = l5 or pc(S,P) = l6) = false .
```

inv42 can be used as a lemma to discharge the three sub-cases. While proving inv42, we encounter three sub-cases of the induction case stlnx. One of the three sub-cases has the following open-close fragment:

```
open MCS-INV .
  eq pc(s,t) = l11 .
  eq next(s,t) = r .
  eq next(s,r) = q .
  eq glock(s) = q .
  eq pc(s,p) = l3 . eq pc(s,q) = l6 . eq pc(s,r) = l6 .
  eq lock(s,q) = true . eq lock(s,r) = true .
  eq (p = t) = false . eq (q = t) = false . eq (r = t) = false .
  eq (p = r) = false . eq (q = r) = false . eq (p = q) = false .
  red inv42(s,p,q) implies inv42(stlnx(s,r),p,q) .
close
```

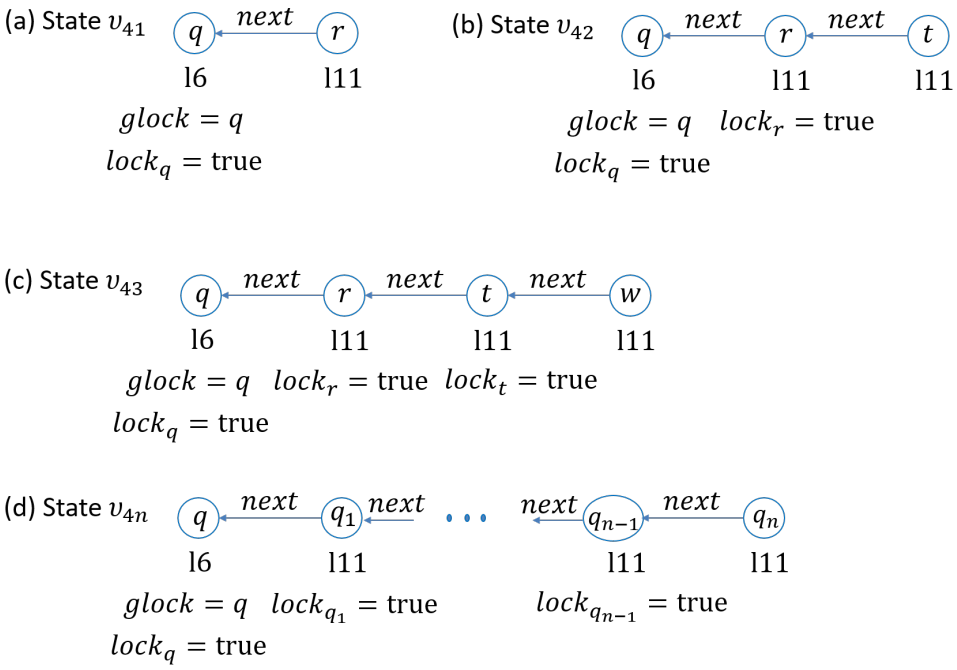


Figure 4.3: States v_{41}, v_{42}, v_{43} & v_{4n}

The other two sub-cases are characterized by almost the same equations. The only difference is `pc(s,p) = l3`, instead of which `pc(s,p) = l4` and `pc(s,p) = l5` hold for the other two sub-cases, respectively. Let v_{42} be an arbitrary state that corresponds to any of the three sub-cases. CafeInMaude returns `false` for all three sub-cases, and then we need a lemma to discharge the three sub-cases.

What if we keep on doing the proof attempt as we did? Let us partially visualize v_{41} as shown in Fig. 4.3 (a). Fig. 4.3 (a) visually says that `q` is located at l6, `r` is located at l11, $next_{\texttt{r}}$ is `q`, $lock_{\texttt{q}}$ is true and *glock* is `q`. Let us partially visualize v_{42} as shown in Fig. 4.3 (b). The difference between v_{41} and v_{42} can be visually observed from Fig. 4.3 (a) and (b). One process located at l6 is inserted between the two processes in Fig. 4.3 (a) and its *lock* is true, although `t` is used in Fig. 4.3 (b) instead of `r` in Fig. 4.3 (a). If we conjecture a lemma, say `inv43`, that can be used to discharge the three sub-cases that correspond to v_{42} as we conjecture `inv41` and `inv42`, we encounter some sub-cases in which `inv43(s,p,q,r,t) implies inv42(stlnx(s,w),p,q,r,t)` reduces to `false` while proving `inv43`. Let v_{43} be an arbitrary state that corresponds to any of the sub-cases. Fig. 4.3 (c) shows the diagram that partially visualizes v_{43}. The difference between Fig. 4.3 (b) and (c) is essentially the same as that of Fig. 4.3 (a) and (b). One more process located at l6

such that its *lock* is true is inserted into the structure constructed with *next* variables. The structure virtually forms the queue in which processes that want to enter the critical section wait. If we repeat what we did, we will encounter the situation that can be partially visualized as shown in Fig. 4.3 (d), which suggests that this way to conjecture lemmas never converges.

There must be a generic lemma that is stronger than `inv41`, `inv42`, etc., but we could not construct such a generic one. Instead, we made `inv40` weaker, constructing `inv4`. By observing some graphical animations of MCS, we realized that there exists at most one process p except for processes q such that (1) q is located at l3 and $prede_q$ is not nop and (2) q is located at l6 and $lock_q$ is true in extended CS region, where extended CS region consists of cs, l7, l8, l10, l11, l3 and l6 [13]. From this observation, we conjectured that whenever there exists such a process p in extended CS region, the virtual queue at least consists of p as an element, which implies that *glock* is not nop. This is `inv4` that is weaker than `inv40`. Tackling the induction case `chglk` for the proof of `inv4`, we encounter a sub-case in which `inv4(s,p) implies inv4(chglk(s,r),p)` reduces to `false`. The sub-case has the following open-close fragment:

```
open MCS-INV .
  op s : -> Sys . ops p r : -> Pid .
  eq pc(s,r) = l8 . eq (p = r) = false .
  eq glock(s) = r . eq pc(s,p) = l3 .
  eq prede(s,p) = nop .
  red inv4(s,p) implies inv4(chglk(s,r),p) .
close
```

This sub-case is quite similar to the one of `inv40`, which has been presented at the beginning of this section. The only difference between the sub-cases of the `inv4` and `inv40` is the existence of `prede(s,p) = nop` in the sub-case of `inv4`. The sub-case of `inv4` can be discharged by using `inv1` as a lemma. The proof of `inv4` needs `inv3` and `inv7` as lemmas as well. Note that υ_{40} is not only the sub-case in which `inv40(s,p) implies inv40(`t_{MCS}`(s, r),p)` reduces to `false`, where t_{MCS} is a transition of $\mathcal{S}_{\mathrm{MCS}}$, but also there are eight more sub-cases such that the term (or formula) reduces to `false`. The eight more sub-cases are characterized by almost the same equations of υ_{40}, except the only difference is `pc(s,p) = l4`, `pc(s,p) = l5`, `pc(s,p) = l6`, `pc(s,p) = cs`, `pc(s,p) = l7`, `pc(s,p) = l8`, `pc(s,p) = l10` and `pc(s,p) = l11` hold for the eight sub-cases, respectively, instead of `pc(s,p) = l3` in υ_{40}. We need to conjecture new lemmas for the first three sub-cases like υ_{40}, while the latter five sub-cases can be discharged by using `inv7` as a lemma.

We strongly believe that inv40 as well as inv41 and inv42 are invariants wrt $\mathcal{S}_{\mathrm{MCS}}$. We were, however, not able to construct any generic lemma that is stronger than all of inv41, inv42, etc., and therefore we have not successfully completed the proof of inv40. Accordingly, we cannot guarantee that inv40 is actually an invariant wrt $\mathcal{S}_{\mathrm{MCS}}$.

Case 2

Let us consider the lemma inv50 obtained by deleting (P = Q) = false from inv5:

```
eq inv50(S,P,Q) = ((next(S,Q) = P and
  (pc(S,Q) = l12 or pc(S,Q) = l1 or pc(S,Q) = rs) = false) implies
  (pc(S,P) = l6 and lock(S,P) = true and pred(S,P) = Q)) .
```

inv50 is stronger than inv5 or equivalently inv5 is weaker than inv50. Since *next* variables are used to virtually construct a queue of process IDs, we put confidence in that $next_p$ never has p as its value. Therefore, we also strongly believe that inv50 is an invariant wrt $\mathcal{S}_{\mathrm{MCS}}$ if inv5 is an invariant wrt $\mathcal{S}_{\mathrm{MCS}}$. We realized, however, that their proofs are totally different when we tried to prove that inv50 is an invariant wrt $\mathcal{S}_{\mathrm{MCS}}$.

The proof of inv5 only uses inv3 as a lemma and the proof of inv3 only uses inv5 as a lemma. On the other hand, the proof of inv50 requires two more lemmas inv51 and inv53 in addition to inv3. The proof of inv51 requires inv3, inv50, inv52 and inv53 as lemmas. The proof of inv52 requires inv53 as a lemma. inv51, inv52 and inv53 are as follows:

```
eq inv51(S,P,Q) = ((next(S,Q) = nop) = false and (P = Q) = false and
  (pc(S,P) = l1 or pc(S,P) = l12 or pc(S,P) = rs) = false and
  (pc(S,Q) = l1 or pc(S,Q) = l12 or pc(S,Q) = rs) = false)
  implies (next(S,P) = next(S,Q)) = false .

eq inv52(S,P) = ((next(S,P) = P) = false) .

eq inv53(S,P) = ((prede(S,P) = P) = false) .
```

Since we suppose that each of $next_p$ and $prede_p$ for every process p is initially set to nop, $next_p$ variables are used to virtually construct a queue and $prede_p$ variables are used to enqueue process IDs as elements into the virtual queue, we are sure that $next_p$ never has p as its value and $prede_p$ never has p as its value. We realized, however, that it is not that straightforward to prove inv53. The proof of inv53 requires a new lemma inv54 whose proof needs five more new lemmas inv55, inv56, inv57, inv58 and inv59 to complete

the proof of inv53. inv54, inv55, inv56, inv57, inv58 and inv59 can be found at the webpage presented in Chapter 1. Let us repeat again that 59 in the notation inv59 does not mean that there are 59 invariants that have been conjectured while conducting formal verification of MCS.

4.5 Formal verification with CiMPA

This section presents how to write proof scripts for CiMPA to formally prove that MCS protocol enjoys the mutual exclusion property. Since the complete proof is quite complicated (consists of more than 2300 lines of code), we only partially describe the proof for a sub-goal, which corresponds to case (8) in Sect. 4.4.

The proof first starts by introducing the goals to prove for CiMPA with the command :goal as follows:

```
open MCS-INV .
  :goal{
    eq [inv1 :nonexec] : inv1(S:Sys,P:Pid,Q:Pid) = true .
    eq [inv2 :nonexec] : inv2(S:Sys,P:Pid,Q:Pid) = true .
    eq [inv3 :nonexec] : inv3(S:Sys,P:Pid) = true .
    eq [inv4 :nonexec] : inv4(S:Sys,P:Pid) = true .
    eq [inv5 :nonexec] : inv5(S:Sys,P:Pid,Q:Pid) = true .
    eq [inv6 :nonexec] : inv6(S:Sys,P:Pid,Q:Pid) = true .
    eq [inv7 :nonexec] : inv7(S:Sys,P:Pid,Q:Pid) = true .
    eq [mutex :nonexec] : mutex(S:Sys,P:Pid,Q:Pid) = true .
  }
```

Let us repeat again that :nonexec indicates that nine equations must not be used for reduction. Then, we can apply induction on S:Sys by using:

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

The commands start the proof by simultaneous induction on S, generating fifteen sub-goals corresponding to fifteen constructors (init and fourteen transitions) in alphabetical order. Each of fifteen sub-goals consists of eight equations corresponding to eight invariants, which are needed to prove. Since the complete proof is quite long, we skip the list of commands that discharge the first sub-goal for chglk. We only partially describe how to discharge the second sub-goal for chlck.

After applying the theorem of constants, let us consider the following sub-goal:

```
2-8. TC
  eq [mutex :nonexec]: mutex(chlck(S#Sys,P#Pid),P@Pid,Q@Pid) = true .
```

`S#Sys` and `P#Pid` are fresh constants introduced by `:apply(si)`, while `P@Pid` and `Q@Pid` are fresh constants introduced by `:apply(tc)`. Goal `2-8` corresponds to case (8) in the proof `mutex` wrt $\mathcal{S}_{\text{MCS}}$ as presented in Sect. 4.4. Similar to the proof score approach, in order to discharge goal `2-8`, it is also neccessary to conduct case splittings, which are similar to those in Table 4.3. We introduce the following commands:

```
:def csb2 = :ctf {eq pc(S#Sys,P#Pid) = l6 .}
:def csb2_1 = :ctf {eq P@Pid = P#Pid .}
:def csb2_2 = :ctf {eq Q@Pid = P#Pid .}
:apply(csb2)
:apply(csb2_1)
:apply(csb2_2)
```

Based on three equations, four sub-goals are generated:

- `2-8-1-1-1` corresponds to case (8.1.1.1) in Table 4.3,
- `2-8-1-1-2` corresponds to three cases (8.1.1.2.*) in Table 4.3,
- `2-8-1-2` corresponds to four cases (8.1.2.*) in Table 4.3,
- `2-8-2` corresponds to case (8.2) in Table 4.3.

Sub-goal `2-8-1-1-1` can be discharged by the following commands:

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

The sub-goal is discharged in this case. The current goal now changes to `2-8-1-1-2`. To discharge `2-8-1-1-2`, we need to conduct case splitting again. The following commands are introduced:

```
:def csb2_3 = :ctf [ lock(S#Sys,P#Pid) .]
:apply(csb2_3)
:imp [mutex] by {P:Pid <- P@Pid ; Q:Pid <- Q@Pid ;}
:apply (rd)
:def csb2_4 = :ctf {eq pc(S#Sys,Q@Pid) = cs .}
:apply(csb2_4)

:imp [inv6] by {P:Pid <- Q@Pid ; Q:Pid <- P@Pid ;}
:imp [mutex] by {P:Pid <- P@Pid ; Q:Pid <- Q@Pid ;}
:apply (rd)
:imp [mutex] by {P:Pid <- P@Pid ; Q:Pid <- Q@Pid ;}
:apply (rd)
```

Based on `csb2_3` and `csb2_4`, `2-8-1-1-2` is split into three sub-goals that correspond to (8.1.1.2.1), (8.1.1.2.2.1) and (8.1.1.2.2.2) in Table 4.3. It requires to use invariant `inv6` as a lemma to discharge the second sub-goal.

The remaining goals can be discharged likewise. It took about 27 minutes to run the proof scripts with CiMPA so as to formally verify that MCS protocol enjoys the mutual exclusion property.

4.6 Formal verification with CiMPG

As explained in the A-Anderson case study, after conducting formal verification by writing proof scores, we can automatically confirm the correctness of the proof scores by using CiMPG to generate proof scripts for CiMPA. Note that, auxiliary lemmas are required in all three ways of formal verification. This is the reason why lemma conjecture is considered as one of the most intellectual/challenging tasks in theorem proving, and LW technique is helpful in formal verification of MCS.

What we need to do are exactly the same as what have been presented in Sect. 3.6. Firstly, in each open-close fragement of the proof scores, we need to add an anotation, e.g., `:id(mcs)`. Then, we ask CiMPG to generate the proof scripts by adding one more open-close fragment as follows:

```
open MCS-INV .
  :proof(mcs)
close
```

Feeding these annotated proof scores into CiMPG, CiMPG generates the proof scripts for CiMPA. Feeding the generated proof scripts into CiMPA, CiMPA discharges all goals, confirming that the proof scores are correct.

In MCS case study, it took about 13 hours and 40 minutes to generate the proof scripts with CiMPG. This may be due to the huge proof scores as the input. Thus, one piece of our future work is to reduce the time of CiMPG in generating proof scripts.

In conclusion, we can summarize what has been presented in this chapter as follows:

- Three ways of formal verification that MCS protocol enjoys the mutual exclusion property. Similar to the A-Anderson case study in Chapter 3, each of the three verification techniques has advantages as well as disadvantages. That is the same point. However, in the MCS case study, the time for CiMPG to generate proof scripts from proof scores is much longer than that of the A-Anderson case study.

- The Lemma Weakening (LW) technique. LW replaces a lemma q_i with q_i' such that $q_i(s)$ implies $q_i'(s)$ for all state s of a state machine M. Auxiliary lemmas are required in all three ways of formal verifications. However, without using LW, we could not have conjectured some good enough lemmas to complete the formal verification of MCS. Accordingly, we would not have been able to complete the formal verification of MCS without the use of LW.

Chapter 5

Releated work

The details of the implementation of CafeInMaude has been presented in paper [5]. In that paper, Riesco, et al. have shown the improvement of the performance of some commands through several case studies with some CafeOBJ specifications. The experiments illustrate that some analyses that could not be performed in CafeOBJ become possible with CafeInMaude. In our experience, it always took less time to execute proof scores with CafeInMaude than that with CafeOBJ. Based on CafeInMaude, Riesco, et al. [7] have continued to develop two extension tools CiMPA and CiMPG. Paper [7] has presented the algorithms behind CiMPA and CiMPG. Qlock protocol, which is an abstract version of Dijkstra's binary semaphore, has been used as a case study to illustrate how to use CiMPA and CiMPG. The paper has also shown the performance of CiMPG in generating proof scripts from the correct proof scores for some protocols.

Anderson protocol has been formally specified in CafeOBJ and semi-formally verified with CafeOBJ [14]. 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.

Wang [15] has proved that it is impossible to automatically prove that concurrent software systems in which multiple processes run algorithms on data structures with pointers enjoy desired properties if there are an arbitrary number of processes. Then, a new approximation method has been proposed to formally verify such software systems. The key idea is to construct a finite collective image set (CIS) whose elements are reachable state representations (or global data-structure image - GDSI). The verification can be done by enumerating all GDSIs reachable from the initial state. He has used the proposed method to prove that a revised version of MCS protocol

enjoys desired properties. The proofs described in the paper, however, are in Mathematical argumentation but not formal. It would at least not be straightforward to develop a tool that fully supports his verification technique.

Rushby [9] has demonstrated that use of disjunctive invariants $q_1 \vee \ldots \vee q_n$ makes invariant verification easier for synchronous concurrent (and/or distributed) systems. His technique proves that $p \wedge (q_1 \vee \ldots \vee q_n)$ is an inductive invariant wrt a system so as to prove that p is an invariant wrt the system. LW, together with LS, can be regarded as a generalized version of his technique. Instead of $p \wedge q_1 \wedge \ldots \wedge q_i \wedge \ldots \wedge q_n$, we prove that $p \wedge q_1 \wedge \ldots \wedge q'_i \wedge \ldots \wedge q_{n'}$ is an inductive invariant wrt a system, where q'_i is weaker than q_i (and n' is much less than n in our case study). q'_i may be in the form $q'_{i1} \vee \ldots \vee q'_{im}$. We have demonstrated that LW can be effective for asynchronous concurrent (and/or distributed) systems as well.

Kim, et al. [16] have used the methodology of certified concurrent abstraction layers to conduct a case study in which they prove that MCS enjoys the lockout freedom property (a liveness property) as well as the mutual exclusion property (a safety property). They have defined five layers such that the lowest one is the implementation of MCS in C/assembly languages and the higher ones are more abstract than the implementation. They have formally proved with Coq, a proof assistant, that each layer except for the highest one contextually refines the one-step higher layer. Their paper mainly focuses on their contextual refinement approach to integration of the verified algorithm, such as MCS, into a larger system, such as an OS.

To prove that p is an invariant wrt $\mathcal{S}$, our method tries to find an inductive invariant q wrt $\mathcal{S}$ such that $q(\upsilon) \Rightarrow p(\upsilon)$ for all states $\upsilon \in \Upsilon$. The model checking algorithm IC3 [17] also can be used for discovering the inductive invariants. Given a finite-state transition system S and a property P that we want to check whether P is invariant for the system S or not, IC3 will gradually refine P, eventually producing either an inductive strengthening of P or a counterexample trace. However, IC3 can not be used to check that the state machine formalizes MCS satisfies the mutual exclusion property or not. The reason is that IC3 only can accept finite-state systems, can not deal with infinite-state systems such as the state machine formalizes MCS. That is the disadvantage of any model checking techniques/tools that we mentioned at the very beginning of the thesis. Our method presented in this paper bases on theorem proving that can get rid of this disadvantage.

Ogata and Futatsugi [18] have reported on a case study in which they have *semi-formally* (but not formally) verified that MCS enjoys the mutual exclusion property and the lockout freedom property in CafeOBJ, although they claimed that their verification is formal. Since their proofs are semi

formal, however, they may have overlooked several subtle cases and then do not discuss anything about what we have encountered. Tam and Ogata [19] have used MCS as a case study to demonstrate the usefulness of their tool called SMGA in helping humans perceive characteristics (or properties) of the state machine formalizes the protocol. Such characteristics have also confirmed by Maude model checking. However, model checking can only guarantee that the characteristics are correct with a fixed and often small number of processes but normally can not do so with an infinite or even a big number of processes due to state explosion. Theorem proving can get rid of this disadvantage.

In the paper [20], the simulation-based verification for invariant properties in the OTS/CafeOBJ method has been proposed. Alternating Bit Protocol (ABP), a communication protocol, together with two more abstract protocols are used to illustrate the method. However, 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. In contrast, we found that the simulation-based verification technique is powerful since it helped us successfully verified that Anderson protocol enjoys the mutual exclusion property. However, this thesis does not describe the part in details.

In addition to CafeOBJ or proof score approach, there are several other languages as well as methods or tools that allow us to conduct formal verification. ACL2 [21] is a well-known formal specification and verification language. It has been used successfully in many formal verification problems. However, in comparison with CafeOBJ, ACL2 seems not as flexible as CafeOBJ. For example, ACL2 syntax only allows prefix expression, while CafeOBJ provides mix-fix syntax (prefix, infix, and postfix all are allowed). Leino [22] has developed a language and verifier called Dafny and illustrated its features through a case study. Dafny is dedicated to formal verification of programs, while our research is dedicated to formal verification of designs (or specifications). Bae, et al. [23] have proposed several abstraction techniques, which are essentially based on narrowing technique, collapsing an infinite state space to a finite one. The techniques have been implemented in the Maude system and two model checking case studies have been conducted. In practice, how to use abstraction techniques correctly is a nontrivial task, somewhat likewise the way we need to find generic lemmas in our OTS/proof score method.

Chapter 6

Conclusion

This thesis has presented the formal verifications that A-Anderson and MCS protocols enjoy the mutual exclusion property in three ways: by writing proof scores and executing them with CafeInMaude, with a proof assistant CiMPA, and with a proof generator CiMPG. Through two case studies, we can summarize some lessons learned. Once we have completely conducted formal verification by writing proof scores, it is rather straightforward to write the proof scripts for CiMPA. Although CiMPG can automatically generate the proof scripts for CiMPA from proof scores, it takes time to do so.

The most intellectual task in both verification case studies is lemma conjecture. Auxiliary lemmas are required in all three ways of verification. For each non-trivial invariant proof, we need to gradually conjecture lemmas that are also invariants during the proof. In the second case study with MCS protocol, we have demonstrated the power of LW, which is a lemma conjecture technique. We were not able to complete the formal proof that MCS enjoys the mutual exclusion property without the use of LW. We had stuck in the proof attempt of `inv40` for several months until we came up with `inv4` that is weaker than `inv40`. `inv4`, together with `inv5`, made us successfully complete the formal proof that MCS enjoys the mutual exclusion property. In fact, we cannot always conjecture the best lemma every time we need to use a lemma. The first lemma we construct may be too weak or strong. Therefore, we may need to strengthen or weaken it. Accordingly, it is natural that it is necessary to use LW as well as LS. To the best knowledge of ours, however, LW has been rarely used in formal methods.

One piece of our future work is to come up with a systematic or hopefully automatic technique to conjecture lemmas that can be used to tackle a large class of systems verification problems even though it is a challenging problem. One possible way to do so is based on LW as well as LS. We will consult some systematic and/or automatic ways to use LS, such those used

by Creme [24], an automatic invariant prover for OTSs specified in Maude. We also aim to develop a tool that automatically generates proof scores for formal systems specifications and formal property specifications such that the tool can scale. Another piece of our future work is to prepare a gentle guide for non-experts to writing proof scripts for CiMPA from their experiences of writing proof scores. Finally, we also aim to come up with better annotations to proof scores for CiMPG to more efficiently generate the proof scripts from annotated proof scores.

List of Publications

[1] Duong Dinh Tran and Kazuhiro Ogata. Formal verification of an abstract version of Anderson protocol with CafeOBJ, CiMPA and CiMPG. In *The 32nd International Conference on Software Engineering and Knowledge Engineering, SEKE 2020, KSIR Virtual Conference Center, Pittsburgh, USA, July 9-19, 2020*, pages 287–292. KSI Research Inc. and Knowledge Systems Institute Graduate School, 2020.

[2] Duong Dinh Tran, Dang Duy Bui, Parth Gupta, and Kazuhiro Ogata. Lemma Weakening for state machine invariant proofs. In *Submitted for publication*, 2020.

Bibliography

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

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

[3] ——, "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. [Online]. Available: https://www.worldscientific.com/doi/abs/10.1142/3831

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

[6] M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and C. Talcott, *All about Maude - a High-Performance Logical Framework: How to Specify, Program and Verify Systems in Rewriting Logic*, ser. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer-Verlag, 2007, vol. 4350.

[7] 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.

[8] 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.

[9] J. M. Rushby, "Verification diagrams revisited: Disjunctive invariants for easy verification," in *CAV*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2000, pp. 508–520.

[10] K. Ogata and K. Futatsugi, "Proof score approach to verification of liveness properties," *IEICE Transactions on Information and Systems*, vol. E91-D, 01 2008.

[11] Z. Manna and A. Pnueli, *Temporal verification of reactive systems - safety*. Berlin, Heidelberg: Springer-Verlag, 1995.

[12] K. Ogata and K. Futatsugi, "A combination of forward and backward reachability analysis methods," in *Formal Methods and Software Engineering*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 501–517.

[13] D. D. Bui and K. Ogata, "Better state pictures facilitating state machine characteristic conjecture," in *26th DMSVIVA*, 2020, pp. 7–12.

[14] 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.

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

[16] J. Kim, V. Sjöberg, R. Gu, and Z. Shao, "Safety and liveness of MCS lock - layer by layer," in *Programming Languages and Systems*. Cham: Springer International Publishing, 2017, pp. 273–297.

[17] A. R. Bradley, "SAT-Based model checking without unrolling," in *Verification, Model Checking, and Abstract Interpretation*, R. Jhala and D. Schmidt, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 70–87.

[18] K. Ogata and K. Futatsugi, "Formal verification of the MCS list-based queuing lock," in *Proceedings of the 5th Asian Computing Science Conference on Advances in Computing Science*, ser. ASIAN '99. Berlin, Heidelberg: Springer-Verlag, 1999, p. 281–293.

[19] T. Nguyen and K. Ogata, *Graphically Perceiving Characteristics of the MCS Lock and Model Checking Them*. Springer International Publishing, 01 2018, pp. 3–23.

[20] 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.

[21] S. Ray, *Scalable Techniques for Formal Verification*, 01 2010.

[22] K. R. M. Leino, "Dafny: An automatic program verifier for functional correctness," in *International Conference on Logic for Programming Artificial Intelligence and Reasoning*, 2010.

[23] K. Bae, S. Escobar, and J. Meseguer, "Abstract logical model checking of infinite-state systems using narrowing," *Leibniz International Proceedings in Informatics, LIPIcs*, vol. 21, pp. 81–96, 01 2013.

[24] M. Nakano, K. Ogata, M. Nakamura, and K. Futatsugi, "Crème: an automatic invariant prover of behavioral specifications," *IJSEKE*, vol. 17, no. 6, pp. 783–804, 2007.