ホテル

佐原伸
（株）CSK システムズ
通信グループ
VDM 担当

2006 年 10 月 5 日

概要

参考文献 [1] 付録 E のホテル部屋の鍵掛けの例題の欠陥を修正し、日本語化し、実行可能仕様とした。

目次

1　問題の概要

ホテルの客は、チェックイン時に、フロントで、部屋の鍵となるカードを受け取る。

カードは 2 つの鍵を持ち、この鍵はその後変わることはない。

フロントは、これまでに発行したカードと、使用済みのカードを記録している。

客が、カードを使って部屋に入ると、前の客のカードでは部屋に入れなくなる。

客は、再度チェックインすることで、新たな鍵となるカードを受け取ることができるが、新しいカードを使うと、古いカードでは部屋に入れなくなる。

1.1 Hotel

ホテルの状態を、状態変数 Hotel で表す。

発行済の「鍵」の集合（発行済）と、どの部屋にどの鍵が使用されたかの情報（使用済）は、フロントが管理している。どの部屋にどの鍵が対応しているかは、「部屋」から「鍵」への写像（部屋の鍵）が記録している。どの客がどのカードを持っているかは、「客」から 「カード」集合への写像（客）が記録している。新しい鍵の候補は、keys が持っている。

参考文献 [1] では、新しい鍵を割り当てるために型束縛を使っているが、仕様を実行可能とするため、本仕様では、新しい鍵は 1 から 9999 までの値を取るようにして、集合束縛により実行が可能なようにした。この値を、あまり大きくすると、仕様実行の際、初期設定で時間がかかってしまう。

もちろん、「鍵」や 「部屋」の型を nat にしているのは、実装のための指定ではない。型の詳細をまだ決めたくない場合、一般的には token 型を使うのだが、テストケースの記述が長くなるのを避けるため、便宜上、nat 型を使っているだけである。

```
module 『ホテル』
definitions
types
「鍵」 = nat;
  「部屋」= nat;
  「客」=token;

  「カード」::
    第一 : 「鍵」
    第二 : 「鍵」;
「フロント」 ::
    発行済 : set of 「鍵」
    使用済 : map 「部屋」 to 「鍵」
    inv f == rng f.使用済 subset f.発行済;

state Hotel of
    フロント : 「フロント」
    部屋の鍵 : map 「部屋」 to 「鍵」
    客 : map 「客」 to set of 「カード」
    keys : set of nat
inv h ==
    dom h.フロント.使用済 subset dom h.部屋の鍵 and
    dunion {{c.第一, c.第二} | c in set dunion rng h.客} subset h.フロント.発行済
init
```

```
   h ==
      h =
         mk_Hotel(
            mk_「フロント」({}, {|->}),
            {|->},
            {|->},
            {1,...,9999}
         )
end
```

1.1.1 チェックイン

新しい鍵を持つ新カードを発行し、その鍵を発行済・使用済の情報に記録する。新カードの情報を、客写像に追加する。

```
operations
チェックイン : 「客」* 「部屋」 ==> 「カード」
チェックイン(guest, room) ==
   let   新鍵  in set keys be st 新鍵 not in set フロント.発行済 in
   let
      新カード = mk_「カード」(フロント.使用済(room), 新鍵)
   in (
      フロント.発行済 := フロント.発行済 union {新鍵};
      フロント.使用済 := フロント.使用済 ++ {room |-> 新鍵};
      客 :=
         if guest in set dom 客 then
            客 ++ {guest |-> 客(guest) union {新カード}}
         else
            客 munion {guest |->  {新カード}};
      return 新カード
   )
pre
   room in set dom フロント.使用済
post
   exists 新鍵 in set keys &
      新鍵 not in set フロント~.発行済 and
      let
         新カード = mk_「カード」(フロント~.使用済(room), 新鍵) in
         フロント.発行済 = フロント~.発行済 union {新鍵} and
```

```
      フロント.使用済 = フロント~.使用済 ++ {room |-> 新鍵} and
      if guest in set dom 客~ then
         客 = 客~ ++ {guest |-> 客~(guest) union {新カード}}
      else
         客 = 客~ munion {guest |->  {新カード}};
```

1.1.2 チェックイン陰仕様

```
チェックイン陰仕様 :「客」* 「部屋」 ==> ()
チェックイン陰仕様(guest, room) == is not yet specified
pre
   room in set dom フロント.使用済
post
   exists 新鍵 :「鍵」 &
      新鍵 not in set フロント~.発行済 and
      let
         新カード = mk_「カード」(フロント~.使用済(room), 新鍵) in
         フロント.発行済 = フロント~.発行済 union {新鍵} and
         フロント.使用済 = フロント~.使用済 ++ {room |-> 新鍵} and
         if guest in set dom 客~ then
            客 = 客~ ++ {guest |-> 客~(guest) union {新カード}}
         else
            客 = 客~ munion {guest |->  {新カード}};
```

1.1.3 入る

客が部屋にはいると、部屋の鍵がカードの第二の鍵になるので、前の客の鍵は使えなくなる。

```
入る :「部屋」*「客」*「カード」==> bool
入る(room, guest, c) == (
   if c.第一 = 部屋の鍵(room) then (
      部屋の鍵 := 部屋の鍵 ++ {room |-> c.第二};
      return true
      )
   elseif c.第二 = 部屋の鍵(room) then
      return true
   else
```

```
      return false
   )
pre
   room in set dom 部屋の鍵 and guest in set dom 客 and
   c in set 客(guest)
post
   if c in set 客(guest) and
      (c.第一 = 部屋の鍵~(room) and 部屋の鍵 = 部屋の鍵~ ++ {room |-> c.第二} or
      c.第二 = 部屋の鍵~(room) and 部屋の鍵 = 部屋の鍵~ )
   then
      RESULT = true
   else
      RESULT = false
   ;
```

1.1.4 入る陰仕様

```
入る陰仕様 :「部屋」*「客」 ==> bool
入る陰仕様(room, guest) == is not yet specified
pre
   room in set dom 部屋の鍵 and guest in set dom 客 and
   exists c in set 客(guest) & c.第一 = 部屋の鍵(room) or c.第二 = 部屋の鍵(room)
post
   if exists c in set 客(guest) &
      c.第一 = 部屋の鍵~(room) and 部屋の鍵 = 部屋の鍵~ ++ {room |-> c.第二} or
      c.第二 = 部屋の鍵~(room) and 部屋の鍵 = 部屋の鍵~
   then
      RESULT =  true
   else
      RESULT = false;
```

1.1.5 ホテル状態を設定する

```
ホテル状態を設定する :
    「フロント」* map 「部屋」 to 「鍵」 * map 「客」 to set of 「カード」 ==> Hotel
ホテル状態を設定する(a フロント, 部屋の鍵写像, 客写像) == (
```

```
   atomic (
      フロント := a フロント;
      部屋の鍵 := 部屋の鍵写像;
      客 := 客写像
   );
   return Hotel
);
```

1.1.6 ホテル状態を得る

```
ホテル状態を得る : () ==> Hotel
ホテル状態を得る() == return Hotel;

end 『ホテル』
```

---

vdm.tc[『ホテル』]

1.2 Hotel の回帰テストケース

module Hotel の回帰テストケース。

t1(), t2(), ... などの各テストケースが返す TestDriver‘TestCase 型の 2 番目の引数は、true を返す式でなければならない。

```
module HotelT
imports
   from TestDriver all,
   from 『ホテル』 all

definitions
values
チェックイン =『ホテル』‘チェックイン;
入る =『ホテル』‘入る;
ホテル状態を得る =『ホテル』‘ホテル状態を得る;
ホテル状態を設定する =『ホテル』‘ホテル状態を設定する;

functions
run : () +> bool
```

```
run() ==
let   testcases = [ t1(), t2(), t3(), t4(), t5(), t6()]
in
TestDriver‘run(testcases);
```

1.2.1 初めてのチェックイン

```
operations
t1 : () ==> TestDriver‘TestCase
t1() ==
   let
      key1 = 1, key2 = 2, key3 = 3,
      room1 =   101, room2 = 102,
      佐原 = mk_token("佐原"),
      oldhotel =
         ホテル状態を設定する(
            mk_『ホテル』‘「フロント」({key1, key2, key3}, {room2 |-> key2}),
            {room1 |-> key1, room2 |-> key2},
            {|->}
         ),
      - = チェックイン(佐原, room2),
      hotel = ホテル状態を得る()
   in
   return
      mk_TestDriver‘TestCase(
      "HotelT01:\t  初めてのチェックイン",
      exists 新鍵  in set oldhotel.keys &
         新鍵 not in set oldhotel.フロント.発行済 and
         let
            新カード = mk_『ホテル』‘「カード」(oldhotel.フロント.使用済(room2), 新鍵) in
            hotel.フロント.発行済 = oldhotel.フロント.発行済 union {新鍵} and
            hotel.フロント.使用済 = oldhotel.フロント.使用済 ++ {room2 |-> 新鍵} and
            hotel.客 = oldhotel.客 munion {佐原 |->  {新カード}}
   )
   ;
```

1.2.2 2 回目以後のチェックイン

```
t2 : () ==> TestDriver`TestCase
t2() ==
   let
      key1 = 1, key2 = 2, key3 = 3,
      room1 =   101, room2 = 102,
      佐原 = mk_token("佐原"),
      - =
         ホテル状態を設定する (
            mk_『ホテル』`「フロント」({key1, key2, key3}, {room2 |-> key2}),
            {room1 |-> key1, room2 |-> key2},
            {|->}
         ),
      - = チェックイン(佐原, room2),
      oldhotel = ホテル状態を得る (),
      - = チェックイン(佐原, room2),
      hotel = ホテル状態を得る ()
   in
   return
      mk_TestDriver`TestCase(
      "HotelT02:\t  2 回目以後のチェックイン",
      exists 新鍵  in set oldhotel.keys &
         新鍵 not in set oldhotel.フロント.発行済 and
         let
            新カード = mk_『ホテル』`「カード」(oldhotel.フロント.使用済(room2), 新鍵) in
            hotel.フロント.発行済 = oldhotel.フロント.発行済 union { 新鍵 } and
            hotel.フロント.使用済 = oldhotel.フロント.使用済 ++ {room2 |-> 新鍵 } and
            hotel.客 = oldhotel.客 ++ { 佐原 |-> oldhotel.客(佐原) union { 新カード }}
   )
   ;
```

1.2.3 入る (第一鍵に一致)

```
t3 : () ==> TestDriver`TestCase
t3() ==
```

```
   let
      key1 = 1, key2 = 2, key3 = 3,
      room1 =   101, room2 = 102,
      佐原 = mk_token("佐原"),
      - =
         ホテル状態を設定する(
            mk_『ホテル』`「フロント」({key1, key2, key3}, {room2 |-> key2}),
            {room1 |-> key1, room2 |-> key2},
            {|->}
         ),
      c = チェックイン(佐原, room2),
      oldhotel = ホテル状態を得る(),
      - =入る(room2, 佐原, c),
      hotel = ホテル状態を得る()
   in
      return
         mk_TestDriver`TestCase(
         "HotelT03:\t  入る(第一鍵に一致)",
         c in set hotel.客(佐原) and
         (c.第一 = oldhotel.部屋の鍵(room2) and
            hotel.部屋の鍵 = oldhotel.部屋の鍵 ++ {room2 |-> c.第二} or
         c.第二 = oldhotel.部屋の鍵(room2) and
            hotel.部屋の鍵 = oldhotel.部屋の鍵)
   );
```

1.2.4 入る(第二鍵に一致)

```
t4 : () ==> TestDriver`TestCase
t4() ==
   let
      key1 = 1, key2 = 2, key3 = 3,
      room1 =   101, room2 = 102,
      佐原 = mk_token("佐原"),
      - =
         ホテル状態を設定する(
            mk_『ホテル』`「フロント」({key1, key2, key3}, {room2 |-> key2}),
            {room1 |-> key1, room2 |-> key2},
            {|->}
```

```
        ),
    c  = チェックイン(佐原, room2),
    - =入る(room2, 佐原, c),
    oldhotel = ホテル状態を得る(),
    - = 入る(room2, 佐原, c),
    hotel = ホテル状態を得る()
in
    return
        mk_TestDriver`TestCase(
        "HotelT04:\t  入る(第二鍵に一致)",
        c in set hotel.客(佐原) and
        (c.第一 = oldhotel.部屋の鍵(room2) and
            hotel.部屋の鍵 = oldhotel.部屋の鍵 ++ {room2 |-> c.第二} or
        c.第二 = oldhotel.部屋の鍵(room2) and
            hotel.部屋の鍵 = oldhotel.部屋の鍵)
);
```

1.2.5 前の客のカードでは入れない。

```
t5 : () ==> TestDriver`TestCase
t5() ==
    let
        key1 = 1, key2 = 2, key3 = 3,
        room1 =   101, room2 = 102,
        佐原 = mk_token("佐原"),
        酒匂 = mk_token("酒匂"),
        - =
            ホテル状態を設定する(
                mk_『ホテル』`「フロント」({key1, key2, key3}, {room2 |-> key2}),
                {room1 |-> key1, room2 |-> key2},
                {|->}
            ),
        c  = チェックイン(佐原, room2),
        - =入る(room2, 佐原, c),
        c1  = チェックイン(酒匂, room2),
        - = 入る(room2, 酒匂, c1),
        - = ホテル状態を得る(),
        err = 入る(room2, 佐原, c)
```

```
   in
      return
         mk_TestDriver`TestCase(
         "HotelT05:\t  前の客のカードでは入れない。",
         not err
   );
```

1.2.6 自分自身でも、前のカードでは入れないが、新しいカードでは入れる。

```
t6 : () ==> TestDriver`TestCase
t6() ==
   let
      key1 = 1, key2 = 2, key3 = 3,
      room1 =   101, room2 = 102,
      佐原 = mk_token("佐原"),
      - =
         ホテル状態を設定する(
            mk_『ホテル』`「フロント」({key1, key2, key3}, {room2 |-> key2}),
            {room1 |-> key1, room2 |-> key2},
            {|->}
         ),
      c  = チェックイン(佐原, room2),
      - =入る(room2, 佐原, c),
      c1  = チェックイン(佐原, room2),
      - = 入る(room2, 佐原, c1),
      - = ホテル状態を得る(),
      oldcard = 入る(room2, 佐原, c),
          newcard = 入る(room2, 佐原, c1)
   in
      return
         mk_TestDriver`TestCase(
         "HotelT05:\t  自分自身でも、前のカードでは入れないが、新しいカードでは入れる。",
         not oldcard and newcard
   );

end HotelT
```

---

vdm.tc[HotelT]

1.3 TestDriver

回帰テストを実行するモジュール。
TestCase 型は、テストケース１件を表す。

```
--£Id: TestDriver.vpp,v 1.1 2005/10/31 02:09:59 vdmtools Exp £
module TestDriver
imports from
   TestLogger all

exports all

definitions
types
TestCase ::
   testCaseName : seq of char
   testResult : bool;

functions
```

---

run は、与えられたテストケース列から結果列を得る。結果がすべて true ならば全体成功メッセージを表示し、１つでも失敗があれば全体失敗メッセージを表示する。

```
run: seq of TestCase +> bool
run(t) ==
   let   m = "Result-of-testcases.",
      r = [isOK(t(i)) | i in set inds t]
   in
   if  forall i in set inds r & r(i) then
      TestLogger`SuccessAll(m)
   else
      TestLogger`FailureAll(m);
```

---

isOK は、与えられたテストケースのテスト結果を確認し、true ならば成功メッセージを表示し、false ならば失敗メッセージを表示する。

```
isOK: TestCase +> bool
isOK(t) ==
   if GetTestResult(t) then
      TestLogger‘Success(t)
   else
      TestLogger‘Failure(t);
```

GetTestResult は、テスト結果を得る。

```
GetTestResult : TestCase +> bool
GetTestResult(t) == t.testResult;
```

GetTestName は、テスト名を得る。

```
GetTestName: TestCase +> seq of char
GetTestName(t) == t.testCaseName;

end TestDriver
```

[TestDriver]vdm.tc[TestDriver]

1.4 TestLogger

テストのログを管理する関数を提供する。

```
--£Id: TestLogger.vpp,v 1.1 2005/10/31 02:09:59 vdmtools Exp £
module TestLogger
imports
   from IO all,
   from TestDriver
      types TestCase
      functions GetTestName

exports all

definitions
```

```
values
historyFileName = "VDMTESTLOG.TXT";

operations
```

Success は、成功メッセージをファイルに追加し、標準出力に表示し、true を返す。

```
Success: TestDriver‘TestCase ==> bool
Success(t) ==
   let   message = TestDriver‘GetTestName(t)^"\tOK.\n"
   in (
      Fprint (message);
      Print(message);
      return true
   );
```

Failure は、失敗メッセージをファイルに追加し、標準出力に表示し、false を返す。

```
Failure: TestDriver‘TestCase ==> bool
Failure(t) ==
   let   message = TestDriver‘GetTestName(t)^"\tNG.\n"
   in (
      Fprint (message);
      Print (message);
      return false
   );
```

SuccessAll は、全体成功メッセージをファイルに追加し、標準出力に表示し、true を返す。

```
SuccessAll : seq of char ==> bool
SuccessAll(m) ==
   let   message = m ^ "\tOK!!\n"
   in (
      Fprint (message);
      Print (message);
      return true
   );
```

FailureAll は、全体失敗メッセージをファイルに追加し、標準出力に表示し、false を返す。

```
FailureAll :  seq of char ==> bool
FailureAll(m) ==
   let   message = m ^ "\tNG!!\n"
   in (
      Fprint (message);
      Print (message);
      return false
   );
```

Print は、標準出力に文字列を表示する。

```
Print : seq of char ==> ()
Print (s) ==
   def - = IO‘echo(s) in skip;
```

Fprint は、現在ディレクトリの historyFileName で示されるファイルに文字列を表示する。

```
operations
Fprint : seq of char ==> ()
Fprint (s) ==
   def - =  IO‘fecho(historyFileName,  s, <append>) in skip;

end TestLogger
```

[TestLogger]vdm.tc[TestLogger]

参考文献

参考文献

[1] Daniel Jackson. *Software Abstraction : Logic, Language, Analysis.* The MIT Press, Cambridge, Massachusetts, London, 2006.