IN-62
190808
71 P

NASA Technical Memorandum 108991

AN ELEMENTARY TUTORIAL ON FORMAL SPECIFICATION AND VERIFICATION USING PVS

Ricky W. Butler

(NASA-TM-108991) AN ELEMENTARY TUTORIAL ON FORMAL SPECIFICATION AND VERIFICATION USING PVS (NASA) 71 p

N94-15536

Unclas

G3/62 0190808

September 1993

National Aeronautics and
Space Administration

Langley Research Center
Hampton, VA 23665

Contents

1 Introduction

This paper carefully guides the reader through the steps of a formal specification and verification of the requirements for a simple system—an airline reservation system. This paper is intended for the novice and is tutorial in nature. The goal is to explore a few important techniques and concepts by way of example rather than to discuss interesting research issues.

This tutorial is intended to be used while one is sitting at a computer terminal. Therefore, general discussions are limited to a few introductory comments. However, the commentary about the example problem is extensive. The reader is referred to [1] for a detailed discussion about contemporary issues in formal methods research.

This tutorial presents the techniques of formal specification and verification in the context of the Prototype Verification System (PVS) developed by SRI International [2]. No specialized knowledge of logic or computer science is assumed, though it is necessary for the reader to have the PVS documentation [3, 4, 5] in order to effectively use this tutorial. The tutorial also assumes that the reader is familiar with Emacs, the text editor the serves as a front-end to the PVS system.

1.1 Some Preliminary Concepts

The requirements specification or high-level design of many systems can be modeled as a state machine. This involves the introduction of an abstract representation of *system state* and a set of operations that operate on the system state. These operations transition a system from one state to another in response to external inputs.

The development of a state machine representation of the system requires the development of a suitable collection of type definitions with which to build the state description. Additional types, constants, and functions are introduced as needed to support subsequent formalization of the operations. Operations on the state are defined as functions that take the system from one state to another or, more generally, as mathematical relations. Many times an *invariant* to the system state is provided to formalize the notion of a "well-defined" system state. The invariant is shown to hold in the presence of an arbitrary operation on the state assuming that the invariant holds before the operation begins. Other desired properties may be expressed as predicates over the system state and operations, and can be proved as *putative theorems* that follow from the formalization.

1.2 Statement of The Example Problem

In the next sections we will demonstrate some of the techniques of formal specification and verification by way of an example—an automated airline seat assignment system that meets the following informal requirements:

1. The system shall make seat assignments for passengers on scheduled airline flights.
2. The system shall maintain a database of seat assignments.
3. The system shall support a fleet having different aircraft types.
4. Passengers shall be allowed to specify preferences for seat type (e.g., window or aisle).
5. The system shall provide the following operations or transactions:
    - Make a new seat assignment
    - Cancel an existing seat assignment

This example problem was derived from an Ehdm specification presented by Ben Di Vito at the Second NASA Formal Methods Workshop [6].

2 Formal Specification of the Reservation System

This section provides a step-by-step elaboration of the process one goes through in developing a formal specification of the example system. Much of the typing required to carry out this exercise can be reduced by retrieving the specifications from `air16.larc.nasa.gov` using anonymous FTP. The specifications are located in the directory `pub/fm/larc/PVS-tutorial` in a file named `plane-reservation-sys.dmp`.

2.1 Creating Basic TYPE Definitions

We begin our formal specification by creating some names for the objects that our formal specification will be describing. We obviously will be talking about seats in an airplane and will need a way to identify a particular seat. We decide to represent an airplane's seating structure as a two-dimensional array of "rows" and "positions". In PVS one writes

```
row:   TYPE
position: TYPE
```

to define the two domains of values. Of course this specification says nothing about what kind of value "row" or "position" could be. We decide to number our rows and positions with positive natural numbers. This is illustrated in figure 1. Of course we really don't need an infinite set of

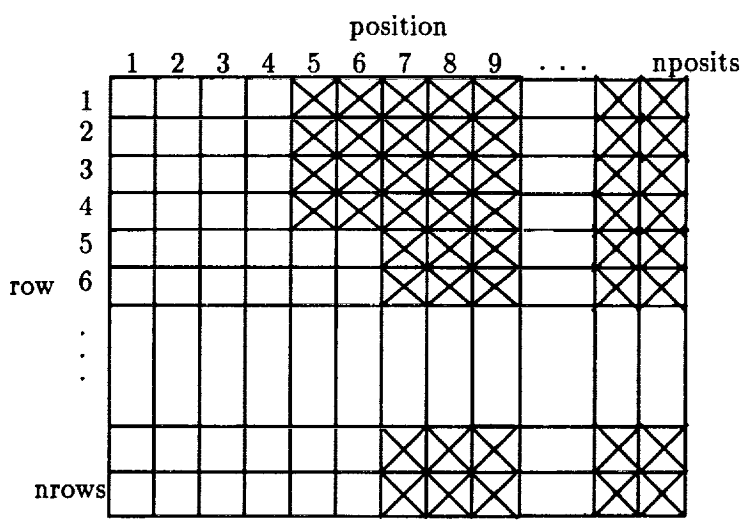

Figure 1: Model of Seating Arrangement In An Airplane

numbers since we know the largest airplane in our fleet, and thus we assume the existence of two constants that delineate the maximum number of rows in any airplane and the maximum number of positions for any row:

```
nrows:  posnat               % Max number of rows
nposits: posnat              % Max number of positions per row
```

We now modify our specification of "row" and "position":

```
row:  TYPE = {n: posnat | 1 <= n AND n <= nrows}
position: TYPE = {n: posnat | 1 <= n AND n <= nposits}
```

This defines **row** and **position** as subranges of the positive naturals (called **posnat** in PVS). The notation is very simple. The text before the | defines the parent type and the text after the | gives a predicate that defines the particular subset of the parent type that you are interested in. Thus, row is any positive natural number between 1 and **nrows** inclusive.

2.2 Creating a PVS Specification File

We now put these together in a file. We start up PVS and type **M-x nf**. PVS asks for a name for the new file. We answer "**basic_defs.pvs**". PVS creates the following file:

```
basic_defs  % [ parameters ]
                : THEORY

  BEGIN

  % ASSUMING
   % assuming declarations
  % ENDASSUMING

  END basic_defs
```

We remove all of the text after the % characters, and then add our type definitions[1]:

```
basic_defs: THEORY
BEGIN
  nrows:  posnat              % Max number of rows
  nposits: posnat             % Max number of positions per row

  row:  TYPE = {n: posnat | 1 <= n AND n <= nrows}
  position: TYPE = {n: posnat | 1 <= n AND n <= nposits}
END basic_defs
```

We now issue the PVS typecheck command, **M-x tc**. PVS responds "**basic_defs typechecked in 0 seconds. No TCCS generated**".

We now need to define some other types that define the flight number, aircraft type, a position preference (e.g. aisle or window) and an identifier for passengers:

```
flight: TYPE                % Flight identifier
plane: TYPE                 % Aircraft type
preference: TYPE            % Position preference
passenger: TYPE             % Passenger identifier
```

We add this text to our file and typecheck again.

[1] The only remaining keywords are **THEORY**, which delineates the start of a new module, and the BEGIN END keywords, which surround the body of the speicification.

2.3 Definition of the Reservation System Database

We are ready to define the database that will maintain all of the reservations. For each flight, the system must maintain a set of seat assignments. We decide to represent each seat assignment as a record that contains a passenger and his assigned seat. This can be formally represented in PVS using the record constructor:

```
seat_assignment: TYPE = [# seat: [row, position],
                           pass: passenger #]
```

The `seat` field of the record is of type `[row, position]`, an ordered pair (or 2-tuple) of `row` and `position`. The entire set of seat assignments for a flight can be represented using PVS's set constructor, `set`.

```
flight_assignments: TYPE = set[seat_assignment]
```

This defines a set type that contains only elements of type `seat_assignment` and assigns it the name `flight_assignments`. Sets are defined in the PVS prelude, which can be displayed using the `M-x vpf` command. The `sets` module provides definitions for the basic set operations. Some of these operations are described in table 1.

operation	traditional notation or meaning
member	$\in$
union	$\cup$
intersection	$\cap$
difference	$\setminus$
add	add element to a set
singleton	constructs set with 1 element
subset?	$\subset$
emptyset	$\emptyset$

Table 1: A partial list of PVS set operations

The complete flight-reservation database can now be modeled as a mapping from flight identifier into that flight's current set of seat assignments:

```
assn_state: TYPE = function[flight -> flight_assignments]
```

Initially, each flight has no assignments:

```
flt: VAR flight
initial_state: function[flight -> flight_assignments] =
               (LAMBDA flt: emptyset[seat_assignment])
```

We add this to our specification and typecheck it.

2.4 Aircraft Seat Layout

Since there is a maximum number of rows and seats per row, we must indicate whether a (`row`, `position`) pair exists for a given aircraft type. This can be accomplished through use of several functions that are uniquely defined for different plane types:

```
seat_exists: function[plane, [row, position]   -> bool]
meets_pref: function[plane, [row, position], preference -> bool]
```

Since we do not want to restrict our specification to any particular plane type, we do not supply a definition (i.e., a function body) for these functions. They are left "uninterpreted." The intended meaning of these functions are as follows. The function **seat_exists** is true only when the indicated seat (i.e. [row,position]) is physically present on the indicated airplane. The function **meets_pref** specifies whether the particular seat is consistent with the particular preference indicated. The type of airplane assigned to a particular flight is given by the **aircraft** function:

```
aircraft: function[flight -> plane]
```

The description of the basic attributes of the system is now complete. The specification is:

```
basic_defs: THEORY
BEGIN

  nrows:  posnat              % Max number of rows
  nposits: posnat             % Max number of positions per row

  row:  TYPE = {n: posnat | 1 <= n AND n <= nrows}
  position: TYPE = {n: posnat | 1 <= n AND n <= nposits}

  flight: TYPE             % Flight identifier
  plane: TYPE              % Aircraft type
  preference: TYPE         % Position preference
  passenger: TYPE          % Passenger identifier

  seat_assignment: TYPE = [# seat: [row, position],
                             pass: passenger #]

  flight_assignments: TYPE = set[seat_assignment]

  assn_state: TYPE = function[flight -> flight_assignments]


  flt: VAR flight
  initial_state: function[flight -> flight_assignments] =
                 (LAMBDA flt: emptyset[seat_assignment])

% ======================================================================
% Definitions that define attributes of a particular airplane
% ======================================================================

  seat_exists: function[plane, [row, position] -> bool]
  meets_pref: function[plane, [row, position], preference -> bool]
  aircraft: function[flight -> plane]

END basic_defs
```

2.5 Specifying Operations on the Database

Our method of formally specifying operations is based on the use of state transition functions. The function defines the value of system state *after* invocation of the operation in terms of the system state *before* the operation is invoked.

To produce a modular specification, we will place the operations in a new theory (i.e. a new module). This is accomplished in PVS by using the **M-x nt** command. We issue this command and name the new theory **ops**. All of the definitions of the **basic_defs** theory are made available to this theory using the **IMPORTING** command:

```
ops: THEORY
BEGIN
  IMPORTING basic_defs
END ops
```

2.6 Seat Assignment Operations

The first operation that we need is `Cancel_assn(flt,pas)`, which cancels the seat assignment for a passenger, **pas**, on flight **flt**:

```
flt:     VAR flight
pas:    VAR passenger
s1:      VAR assn_state
a,b:     VAR seat_assignment
Cancel_assn: function[flight, passenger, assn_state -> assn_state] =
  (LAMBDA flt, pas, s1:
        s1 WITH [(flt) := {a | member(a,s1(flt)) AND pass(a) /= pas}])
```

This specification uses the PVS **WITH** construct. The **WITH** expression is used to define a new function that differs from another function for a few indicated values. For example, f **WITH** [(1) := y] is identical to f, except possibly for f(1)[2]. Thus, all seat assignment sets for flights other than **flt** are unchanged. For flight **flt**, however, all assignments on behalf of passenger **pas** are removed (there should be at most one). As discussed earlier (i.e. see table 1), the function **member** is defined in the **sets** module of the PVS prelude.

The second operation is `Make_assn(flt,pas,pref)`, which makes a seat assignment, if possible, for passenger **pas** on flight **flt**. There are two conditions that should prevent us from carrying out this operation on the reservation database

1. when there is no seat available that meets the passenger's specified preference

2. when the passenger already has a seat on the plane

Condition (1) can be expressed in PVS as follows:

```
(FORALL seat:  meets_pref(aircraft(flt), seat, pref)
        IMPLIES (EXISTS a: member(a, as(flt))
                            AND seat(a) = seat)))
```

[2]The resulting function is not different if f(1) = y.

This states that all seats that meet the passenger's preference (**meets_pref(aircraft(flt), seat, pref)**) are already assigned to another passenger, i.e., there already exists a record **a** in the database with the specified seat. Note that PVS departs from the traditional dot notation (e.g. **a.seat**) and uses **seat(a)** to dereference the seat field of record **a**. We can supply a name, **pref_filled** for this condition as follows:

```
as:    VAR assn_state
pref:  VAR preference
seat:  VAR [row,position]
pref_filled: function[assn_state, flight, preference -> bool] =
  (LAMBDA as, flt, pref:
     (FORALL seat:  meets_pref(aircraft(flt), seat, pref)
            IMPLIES (EXISTS a: member(a, as(flt)) AND seat(a) = seat)))
```

The first line gives the types of the arguments and result of the function. The key word LAMBDA is just syntax that means "the following text up to the colon are the formal arguments for this function. PVS also allows the following equivalent definition:

```
pref_filled(as, flt, pref): bool =
     (FORALL seat:  meets_pref(aircraft(flt), seat, pref)
                IMPLIES (EXISTS a: member(a, as(flt))
                                   AND seat(a) = seat))
```

where the types of the function arguments are inferred from the variable declarations.

The second condition (i.e., the passenger already has a seat on the plane) can be defined as follows:

```
pass_on_flight: function[passenger, flight, assn_state -> bool] =
                (LAMBDA pas, flt, s1: (EXISTS a:
                                  pass(a) = pas AND member(a,s1(flt))))
```

We are now ready to define the operation that assigns a passenger to a particular flight, **Make_assn**:

```
Make_assn: function[flight, passenger, preference,  assn_state
                        -> assn_state] =
  (LAMBDA flt, pas, pref, s1:
      IF pref_filled(s1, flt, pref) OR
         pass_on_flight(pas,flt,s1) THEN s1
      ELSE
         (LET a = (# seat := Next_seat(s1,flt,pref),
                     pass := pas #) IN
            s1 WITH [(flt) := add(a, s1(flt))])
      ENDIF)
```

In this specification, if either of the two anomalous conditions is true, the database is not changed. The ELSE clause defines what happens otherwise. This clause uses PVS's LET construct. The LET statement allows one to assign a name to a subexpression. This is especially useful when a subexpression is used multiple times in an expression. In our case, the subexpression **a** only occurs once—in the subexpression **add(a, s1(flt))**, which creates a new set by adding the element **a** to the set s1(flt). The LET is used here to make the complete expression easier to read. The value of the LET variable a is defined using a record constructor, i.e. (# ... #). In this case, the pass field is set equal to the formal parameter pas, and the seat field of the record is updated with the result from another function, Next_seat:

```
Next_seat: function[assn_state, flight, preference -> [row,position]]
```

This function selects the next seat to be given a passenger from all of the available seats. Any number of algorithms can be imagined that would make this selection, e.g. the seat with the lowest row and position number available. However, since this is a high-level specification, we decide to leave the particular selection algorithm unspecified. Thus, we do not define a body for this function and leave it as an "uninterpreted" function. Nevertheless, we will need a general property about this function in order for one of our proofs to go through[3]. We define this property with an axiom:

```
Next_seat_ax: AXIOM NOT pref_filled(s1, flt, pref) IMPLIES
      seat_exists(aircraft(flt),Next_seat(s1,flt,pref))
```

This axiom states that if a seat is available that matches the specified preference, then the function Next_seat returns a [row, position] that actually exists on the airplane scheduled for flight flt.

Now that we have defined the operations, we are faced with the question, "How do we know that the operations were specified correctly?" One approach to this problem is to construct "putative" theorems. These are properties about the operations that should be true if we have defined them properly. For example,

```
Make_Cancel: THEOREM NOT pass_on_flight(pas,flt,s1) =>
                 Cancel_assn(flt,pas,Make_assn(flt,pas,pref,s1)) = s1
```

This states that if a particular passenger is not already assigned to a flight, then the result of assigning that passenger to a flight and then canceling his reservation will return the database to its original state[4]. The process of attempting to prove such theorems can lead to the discovery of errors in the specification. The proof of this theorem will be given in a later section. Some other examples are:

```
Cancel_putative: THEOREM
        NOT (EXISTS (a: seat_assignment):
           member(a,Cancel_assn(flt,pas,s1)(flt)) AND pass(a) = pas)

Make_putative: THEOREM NOT pref_filled(s1, flt, pref) =>
        (EXISTS (x: seat_assignment):
          member(x, Make_assn(flt, pas, pref, s1)(flt)) AND pass(x) = pas)
```

2.7 Specifying Invariants On the State Of the Database

The system state is subject to three types of anomalies:

1. Assigning nonexistent seats to passengers

2. Assigning multiple seats to a single passenger

[3]The need for this property was not apparent until the proofs were in progress.

[4]We have used the alternate PVS syntax for logical implies: =>.

3. Assigning more than one passenger to a single seat

Prevention of anomaly (1) can be formalized as follows:

```
existence: function[assn_state -> bool] =
  (LAMBDA as: (FORALL a,flt: member(a, as(flt)) IMPLIES
     seat_exists(aircraft(flt), seat(a))))
```

Prevention of anomaly (2) can be formalized as follows:

```
uniqueness: function[assn_state -> bool] =
  (LAMBDA as: (FORALL a,b,flt:
     member(a, as(flt)) AND member(b, as(flt))
         AND pass(a) = pass(b) IMPLIES a = b))
```

Prevention of anomaly (3) can be formalized as follows:

```
one_per_seat: function[assn_state -> bool] =
   (LAMBDA as:  (FORALL a,b,flt: member(a, as(flt)) AND member(b, as(flt))
         AND seat(a) = seat(b) IMPLIES a = b))
```

The overall state invariant is the conjunction of the three. However, in order to simplify the discussion we will work with the first two and leave the last invariant as an exercise[5]. The conjunction of the first two can be captured in a single function as follows:

```
assn_invariant: function[assn_state -> bool] =
  (LAMBDA as: existence(as) AND uniqueness(as))
```

2.8 PVS Typechecking and Typecheck Conditions (TCCs)

We combine the definitions for the operations and the invariants in a new theory called **ops**:

```
ops: THEORY

BEGIN

  IMPORTING basic_defs

  flt:    VAR flight
  pas:   VAR passenger
  as, s1: VAR assn_state
  a,b,x:       VAR seat_assignment
  pref:   VAR preference
  seat:   VAR [row,position]

  Cancel_assn: function[flight, passenger, assn_state -> assn_state] =
    (LAMBDA flt, pas, s1:
          s1 WITH [(flt) := {a | member(a,s1(flt)) AND pass(a) /= pas}])
```

[5] It is the easiest of the three invariants.

```
pref_filled: function[assn_state, flight, preference -> bool] =
  (LAMBDA as, flt, pref:
     (FORALL seat:  meets_pref(aircraft(flt), seat, pref)
                 IMPLIES (EXISTS a: member(a, as(flt))
                                    AND seat(a) = seat)))

Next_seat: function[assn_state, flight, preference -> [row,position]]

Next_seat_ax: AXIOM NOT pref_filled(s1, flt, pref) IMPLIES
      seat_exists(aircraft(flt),Next_seat(s1,flt,pref))

pass_on_flight: function[passenger, flight, assn_state -> bool] =
                (LAMBDA pas, flt, s1: (EXISTS a:
                                   pass(a) = pas AND member(a,s1(flt))))

Make_assn: function[flight, passenger, preference,  assn_state
                      -> assn_state] =
  (LAMBDA flt, pas, pref, s1:
      IF pref_filled(s1, flt, pref) OR
         pass_on_flight(pas,flt,s1) THEN s1
      ELSE
         (LET a = (# seat := Next_seat(s1,flt,pref),
                     pass := pas #) IN
            s1 WITH [(flt) := add(a, s1(flt))])
      ENDIF)

% ======================================================================
%                               Invariants
% ======================================================================

existence: function[assn_state -> bool] =
  (LAMBDA as: (FORALL a,flt: member(a, as(flt)) IMPLIES
                             seat_exists(aircraft(flt), seat(a))))

uniqueness: function[assn_state -> bool] =
  (LAMBDA as: (FORALL a,b,flt:
     member(a, as(flt)) AND member(b, as(flt))
         AND pass(a) = pass(b) IMPLIES a = b))

assn_invariant: function[assn_state -> bool] =
  (LAMBDA as: existence(as) AND uniqueness(as))

Cancel_assn_inv: THEOREM assn_invariant(s1)
                       Implies assn_invariant(Cancel_assn(flt,pas,s1))

MAe: THEOREM existence(s1)
             IMPLIES existence(Make_assn(flt,pas,pref,s1))
```

```
MAu: THEOREM uniqueness(s1)
              IMPLIES uniqueness(Make_assn(flt,pas,pref,s1))
Make_assn_inv: THEOREM assn_invariant(s1) =>
                        assn_invariant(Make_assn(flt,pas,pref,s1))

Make_Cancel: THEOREM NOT pass_on_flight(pas,flt,s1) =>
                 Cancel_assn(flt,pas,Make_assn(flt,pas,pref,s1)) = s1

END ops
```

When we issue the **M-x tc** command we notice that the system responds **ops typechecked: 2 TCCs, 0 Proved, 0 subsumed, 2 unproved.** Unlike many high-level programming languages, PVS often requires theorem proving in order to guarantee that the specification is type correct. This is the price one has to pay for the very powerful type structure of the language.

M-x show-tccs opens up a window that displays the typecheck obligations:

```
% Existence TCC generated for row
  % unproved
Next_seat_TCC1: OBLIGATION (EXISTS (x1: posnat): 1 <= x1 AND x1 <= nrows)

% Existence TCC generated for position
  % unproved
Next_seat_TCC2: OBLIGATION (EXISTS (x1: posnat): 1 <= x1 AND x1 <= nposits)
```

We position the cursor on the first obligation and type **M-x pr**. The PVS system responds by opening up a proof buffer containing the following output:

```
Next_seat_TCC1 :

  |-------
{1}   (EXISTS (x1: posnat): 1 <= x1 AND x1 <= nrows)
```

The system wants us to prove that there exists a positive natural number between 1 and **nrows**. We suggest that number 1 is such a number using the **INST** command.

```
Rule? (inst 1 "1")
Instantiating the top quantifier in 1 with the terms:
 1
this simplifies to:
Next_seat_TCC1 :

  |-------
{1}   1 <= 1 AND 1 <= nrows
```

We then tell the theorem prover that we think we have a proof via the **ASSERT** command:

```
Rule? (assert)
Invoking decision procedures,
```

```
Q.E.D.

Run time  = 1.28 secs.
Real time = 25.15 secs.
NIL
```

The prover agrees and responds Q.E.D. The same two commands prove the other obligation as well. The use of the theorem prover will be explored in more detail in the next section.

There is an alternative way to deal with these TCC obligations—through use of the PVS CONTAINING clause. Both of these proofs only needed the existence of a member of the user-defined subtype. The CONTAINING clause enables one to affirm such a member in the specification:

```
row:  TYPE = {n: posnat | 1 <= n AND n <= nrows} CONTAINING 1
position: TYPE = {n: posnat | 1 <= n AND n <= nposits} CONTAINING 1
```

If row and position are defined in this manner, the generated TCCs are automatically proven using the command M-x tcp.

3 Formal Verifications

In this section we will walk-through the mechanical verification of the invariant properties discussed previously.

To establish that the state invariant is preserved by every operation, we must prove theorems of the form:

$$I(S_1) \supset \mathrm{I}(\mathrm{op\_spec}(S_1))$$

where S_1 is the state before the operation and I represents the state invariant. Note that op_spec(S_1) is the state of the system after the operation. For our example, the required theorems can be expressed as follows:

```
Cancel_assn_inv: THEOREM assn_invariant(s1)
                      Implies assn_invariant(Cancel_assn(flt,pas,s1))

Make_assn_inv: THEOREM assn_invariant(s1) =>
                      assn_invariant(Make_assn(flt,pas,pref,s1))
```

3.1 Proof that Cancel_assn Maintains the Invariant

Although it is almost always advisable to search for a proof before engaging the theorem prover, the prover can be useful in the discovery of the proof. We shall use this approach on this example, since the theorem is shallow and not hard to understand. This section is meant to guide the PVS novice through his first non-trivial use of the theorem prover. The goal is to gain some familiarity with the capabilities of the system so that further reading in the user manuals is more productive. The proofs given in this tutorial are by no means the best way of proving these theorems. They were performed with the goal of walking the user through a large number of the PVS commands.

The user begins a proof session by positioning the cursor on a proof and typing M-x pr. The system responds with:

```
Cancel_assn_inv :

  |-------
{1}   (FORALL (flt: flight), (pas: passenger), (s1: assn_state):
         assn_invariant(s1) IMPLIES assn_invariant(Cancel_assn(flt, pas, s1)))
```

The user then issues commands that manipulate the formula using truth-preserving operations. The goal, of course, is to simplify the formula to the point where the prover can identify the formula as a tautology (i.e. and thus a theorem). The user input in this paper can be identified by the `rule?` prompt. All of the inputs in this tutorial are only one line. Although PVS allows commands to be entered in either lower or upper case, we will use upper-case letters exclusively to enhance readability.

The first thing that one usually does when proving a formula containing quantifiers (i.e. `FORALLS` or `EXISTS`) is to remove them. This is necessary because many of the PVS commands are only effective when the quantifiers have been removed. There are two basic strategies for removing quantifiers: skolemization and quantification. Some situations require skolemization and others require quantification. In this case we need to skolemize formula [1][6]. In PVS this is accomplished using the `SKOLEM` command:

```
Rule? (SKOLEM 1 ("Flt" "Pas" "S1"))
For the top quantifier in *, we introduce Skolem constants: (Flt Pas S1)
this simplifies to:
Cancel_assn_inv :

  |-------
{1}   assn_invariant(S1) IMPLIES assn_invariant(Cancel_assn(Flt, Pas, S1))
```

The first parameter of the `SKOLEM` command (i.e., 1,) specifies which formula the command should be applied to[7]. Note that the list of skolem names are enclosed in parentheses as well as the complete command itself.

Clearly, no progress can be made until the meaning of "assn_invariant" is exposed to the prover. This is done through use of the `EXPAND` command, i.e. `(EXPAND "assn_invariant")`. The system responds as follows:

```
Rule? (EXPAND "assn_invariant")
Expanding the definition of assn_invariant
this simplifies to:
Cancel_assn_inv :

  |-------
{1}   existence(S1) AND uniqueness(S1)
        IMPLIES existence(Cancel_assn(Flt, Pas, S1))
          AND uniqueness(Cancel_assn(Flt, Pas, S1))
```

[6]The basic idea of skolemization is that a formula like $\forall x : P(x)$ which asserts the validity of a predicate P for an arbitrary value of x, is equivalent to $P(a)$ where a is a previously unused constant.

[7]In this case there is only one available formula. Later we shall encounter sequents with several formulas.

We then proceed to expand with definitions of **existence**, **uniqueness** and **Cancel_assn**:

```
Rule? (EXPAND "existence")
Expanding the definition of existence
this simplifies to:
Cancel_assn_inv :

  |-------
{1}   (FORALL (a: seat_assignment), (flt: flight):
        member(a, S1(flt)) IMPLIES seat_exists(aircraft(flt), seat(a)))
       AND uniqueness(S1)
       IMPLIES
       (FORALL (a: seat_assignment), (flt: flight):
          member(a, Cancel_assn(Flt, Pas, S1)(flt))
            IMPLIES seat_exists(aircraft(flt), seat(a)))
         AND uniqueness(Cancel_assn(Flt, Pas, S1))

Rule? (EXPAND "uniqueness")
Expanding the definition of uniqueness
this simplifies to:
Cancel_assn_inv :

  |-------
{1}   (FORALL (a: seat_assignment), (flt: flight):
        member(a, S1(flt)) IMPLIES seat_exists(aircraft(flt), seat(a)))
       AND
       (FORALL (a: seat_assignment), (b: seat_assignment), (flt: flight):
          member(a, S1(flt)) AND member(b, S1(flt)) AND pass(a) = pass(b)
            IMPLIES a = b)
       IMPLIES
       (FORALL (a: seat_assignment), (flt: flight):
          member(a, Cancel_assn(Flt, Pas, S1)(flt))
            IMPLIES seat_exists(aircraft(flt), seat(a)))
         AND
         (FORALL (a: seat_assignment), (b: seat_assignment), (flt: flight):
            member(a, Cancel_assn(Flt, Pas, S1)(flt))
              AND member(b, Cancel_assn(Flt, Pas, S1)(flt))
                AND pass(a) = pass(b)
              IMPLIES a = b)

Rule? (EXPAND "Cancel_assn")
Expanding the definition of Cancel_assn
this simplifies to:
Cancel_assn_inv :

  |-------
{1}   (FORALL (a: seat_assignment), (flt: flight):
        member(a, S1(flt)) IMPLIES seat_exists(aircraft(flt), seat(a)))
```

```
AND
(FORALL (a: seat_assignment), (b: seat_assignment), (flt: flight):
   member(a, S1(flt)) AND member(b, S1(flt)) AND pass(a) = pass(b)
     IMPLIES a = b)
IMPLIES
(FORALL (a: seat_assignment), (flt: flight):
   member(a,
          S1
            WITH [Flt :=
              {a: seat_assignment |
              member(a, S1(Flt)) AND pass(a) /= Pas}](flt))
     IMPLIES seat_exists(aircraft(flt), seat(a)))
  AND
  (FORALL (a: seat_assignment), (b: seat_assignment), (flt: flight):
     member(a,
            S1
              WITH [Flt :=
                {a: seat_assignment |
                member(a, S1(Flt)) AND pass(a) /= Pas}](flt))
       AND
       member(b,
              S1
                WITH [Flt :=
                  {a: seat_assignment |
                  member(a, S1(Flt)) AND pass(a) /= Pas}](flt))
         AND pass(a) = pass(b)
       IMPLIES a = b)
```

We note that the function member appears in several places in the formula. Although this function is defined in the PVS prelude[8], it must still be expanded in order for PVS to know what it means:

```
Rule? (EXPAND "member")
Expanding the definition of member
this simplifies to:
Cancel_assn_inv :

  |-------
{1}   (FORALL (a: seat_assignment), (flt: flight):
        S1(flt)(a) IMPLIES seat_exists(aircraft(flt), seat(a)))
      AND
      (FORALL (a: seat_assignment), (b: seat_assignment), (flt: flight):
         S1(flt)(a) AND S1(flt)(b) AND pass(a) = pass(b) IMPLIES a = b)
      IMPLIES
      (FORALL (a: seat_assignment), (flt: flight):
         S1
           WITH [Flt :=
             {a: seat_assignment | S1(Flt)(a) AND pass(a) /= Pas}](flt)(a)
```

[8]Can be inspected by typing M-x vpf.

```
      IMPLIES seat_exists(aircraft(flt), seat(a)))
   AND
   (FORALL (a: seat_assignment), (b: seat_assignment), (flt: flight):
      S1
        WITH [Flt :=
          {a: seat_assignment | S1(Flt)(a) AND pass(a) /= Pas}](flt)(a)
        AND
        S1
          WITH [Flt :=
            {a: seat_assignment |
            S1(Flt)(a) AND pass(a) /= Pas}](flt)(b)
          AND pass(a) = pass(b)
        IMPLIES a = b)
```

Notice that `member(a,S1(flt))` has been changed to `S1(flt)(a)`. This looks funny at first, but it is correct. In PVS, sets are represented as functions that map from the domain type of the set into boolean. This boolean-valued function is true only for members of the set, i.e. `S(x)` is true if and only if $\mathtt{x} \in \mathtt{S}$.

We are now ready to issue a useful command that breaks up formulas in a sequent into smaller more tractable pieces: **FLATTEN**:

```
Rule? (FLATTEN)
Applying disjunctive simplification to flatten sequent,
this simplifies to:
Cancel_assn_inv :

{-1}   (FORALL (a: seat_assignment), (flt: flight):
         S1(flt)(a) IMPLIES seat_exists(aircraft(flt), seat(a)))
{-2}   (FORALL (a: seat_assignment), (b: seat_assignment), (flt: flight):
         S1(flt)(a) AND S1(flt)(b) AND pass(a) = pass(b) IMPLIES a = b)
  |-------
{1}   (FORALL (a: seat_assignment), (flt: flight):
         S1
           WITH [Flt :=
             {a: seat_assignment | S1(Flt)(a) AND pass(a) /= Pas}](flt)(a)
           IMPLIES seat_exists(aircraft(flt), seat(a)))
        AND
        (FORALL (a: seat_assignment), (b: seat_assignment), (flt: flight):
           S1
             WITH [Flt :=
               {a: seat_assignment | S1(Flt)(a) AND pass(a) /= Pas}](flt)(a)
             AND
             S1
               WITH [Flt :=
                 {a: seat_assignment | S1(Flt)(a) AND pass(a) /= Pas}](flt)(b)
               AND pass(a) = pass(b)
             IMPLIES a = b)
```

This is a probably a good place to discuss in more detail the nature of a "sequent". The system has broken the formula into three separate formulas labeled "{-1}, {-2}" and "{1}" separated by a horizontal line. The basic idea is that all of the formulas labeled with positive numbers logically follow from the formulas labeled with negative numbers. More precisely, the conjunction of the antecedent (i.e. negative) formulas logically implies the disjunction of the consequent (i.e. positive) formulas. In this instance we have:

$$\{-1\} \wedge \{-2\} ==> \{1\}$$

We now notice that formula {1} is the conjunction (i.e. AND) of two formulas. In order for the formula to be true, each of these must separately be true. To reduce the amount of text that we have to think about at one time, it is helpful to break the proof into two separate steps. The PVS system lets us do this with the SPLIT command.

```
Rule? (SPLIT 1)
Splitting conjunctions,
this yields  2 subgoals:
Cancel_assn_inv.1 :

[-1]   (FORALL (a: seat_assignment), (flt: flight):
         S1(flt)(a) IMPLIES seat_exists(aircraft(flt), seat(a)))
[-2]   (FORALL (a: seat_assignment), (b: seat_assignment), (flt: flight):
         S1(flt)(a) AND S1(flt)(b) AND pass(a) = pass(b) IMPLIES a = b)
  |-------
{1}   (FORALL (a: seat_assignment), (flt: flight):
         S1
           WITH [Flt :=
             {a: seat_assignment | S1(Flt)(a) AND pass(a) /= Pas}](flt)(a)
           IMPLIES seat_exists(aircraft(flt), seat(a)))
```

Notice that PVS responds with "`Splitting conjunctions, this yields 2 subgoals:`. We now have two sequents to prove. These are named `Cancel_assn_inv.1` and `Cancel_assn_inv.2`. The system automatically keeps track of what has been proved and what is still unfinished. After we finish proving `Cancel_assn_inv.1` the system will require us to prove `Cancel_assn_inv.2`[9].

Things are starting to look a bit more tractable. Since we still have universal quantifiers in formula {1}, we decide to skolemize it. This time we will use the SKOLEM! command. This tells the theorem prover to use any names that it likes.

```
Rule? (SKOLEM! 1)
For the top quantifier in 1, we introduce Skolem constants: (a!1 flt!1)
this simplifies to:
Cancel_assn_inv.1 :

[-1]   (FORALL (a: seat_assignment), (flt: flight):
         S1(flt)(a) IMPLIES seat_exists(aircraft(flt), seat(a)))
[-2]   (FORALL (a: seat_assignment), (b: seat_assignment), (flt: flight):
         S1(flt)(a) AND S1(flt)(b) AND pass(a) = pass(b) IMPLIES a = b)
  |-------
```

[9]Only one of the subgoals is displayed by the system at a time. The PVS command POSTPONE can be used to switch to another subgoal.

```
{1}   S1
        WITH [Flt :=
          {a: seat_assignment | S1(Flt)(a) AND pass(a) /= Pas}](flt!1)(a!1)
        IMPLIES seat_exists(aircraft(flt!1), seat(a!1))
```

As we can see, the prover chose names a!1 and flt!1. This is not our first choice in names, but at least this approach saved some typing. It also has the advantage that the name of the original quantified variable is easily retrieved from the skolem name.

We notice that formula [-1] almost implies formula {1} (that is, after substituting a!1 and flt!1 for the universal (i.e. FORALL) variables). The only difference is that in formula {1} the function S1 is slightly modified—the value of S1(Flt) has been changed. We would like to deal with this case separately. This is accomplished by using the CASE command:

```
Rule? (CASE "Flt = flt!1")
Case splitting on
   Flt = flt!1,
this yields  2 subgoals:
Cancel_assn_inv.1.1 :

{-1}   Flt = flt!1
[-2]   (FORALL (a: seat_assignment), (flt: flight):
         S1(flt)(a) IMPLIES seat_exists(aircraft(flt), seat(a)))
[-3]   (FORALL (a: seat_assignment), (b: seat_assignment), (flt: flight):
         S1(flt)(a) AND S1(flt)(b) AND pass(a) = pass(b) IMPLIES a = b)
  |-------
[1]   S1
        WITH [Flt :=
          {a: seat_assignment | S1(Flt)(a) AND pass(a) /= Pas}](flt!1)(a!1)
        IMPLIES seat_exists(aircraft(flt!1), seat(a!1))
```

PVS responds with Case splitting on Flt = flt!1, this yields 2 subgoals. In one of the subgoals, Flt = flt!1 is put on the antecedent list and NOT Flt = flt!1 is put on the other list. PVS will actually move the NOT formula to the consequent side and remove the NOT. This is logically equivalent. We now have three sequents to deal with. However, each of these are simpler to prove than the original one.

We now issue the ASSERT command. This command invokes the PVS decision procedures to analyze the sequent. When there are no quantifiers left around and the formulas have been reduced to the point where simple propositional reasoning is adequate, ASSERT will automatically finish off the proof. In this case, we still have quantifiers on the antecedent side of the sequent, and ASSERT does not finish the job. Nevertheless, ASSERT does simplify the sequent for us:

```
Rule? (ASSERT)
Invoking decision procedures,
this simplifies to:
Cancel_assn_inv.1.1 :

[-1]   Flt = flt!1
[-2]   (FORALL (a: seat_assignment), (flt: flight):
         S1(flt)(a) IMPLIES seat_exists(aircraft(flt), seat(a)))
```

```
[-3]   (FORALL (a: seat_assignment), (b: seat_assignment), (flt: flight):
         S1(flt)(a) AND S1(flt)(b) AND pass(a) = pass(b) IMPLIES a = b)
  |-------
{1}   S1(Flt)(a!1) AND pass(a!1) /= Pas
        IMPLIES seat_exists(aircraft(flt!1), seat(a!1))
```

Notice that the WITH structure has been collapsed in formula [1]. As we noticed before, formula [-2] implies formula {1}, but it contains universally quantified (i.e. FORALL) variables that must be instantiated before the PVS decision procedures can effectively work with it. Thus we substitute a!1 and flt!1 for the universal variables in [-2] . This is done in PVS using the INST command:

```
Rule? (INST -2 "a!1" "Flt")
Instantiating the top quantifier in -2 with the terms:
 (a!1 Flt)
this simplifies to:
Cancel_assn_inv.1.1 :

[-1]   Flt = flt!1
{-2}   S1(Flt)(a!1) IMPLIES seat_exists(aircraft(Flt), seat(a!1))
[-3]   (FORALL (a: seat_assignment), (b: seat_assignment), (flt: flight):
         S1(flt)(a) AND S1(flt)(b) AND pass(a) = pass(b) IMPLIES a = b)
  |-------
[1]   S1(Flt)(a!1) AND pass(a!1) /= Pas
        IMPLIES seat_exists(aircraft(flt!1), seat(a!1))
```

Formula [1] directly follows from [-1] and [-2], but we remember that PVS's decision procedures often need formulas that contain an IMPLIES to be flattened. So we issue a FLATTEN command:

```
Rule? (FLATTEN)
Applying disjunctive simplification to flatten sequent,
this simplifies to:
Cancel_assn_inv.1.1 :

[-1]   Flt = flt!1
{-2}   S1(Flt)(a!1)
[-3]   S1(Flt)(a!1) IMPLIES seat_exists(aircraft(Flt), seat(a!1))
[-4]   (FORALL (a: seat_assignment), (b: seat_assignment), (flt: flight):
         S1(flt)(a) AND S1(flt)(b) AND pass(a) = pass(b) IMPLIES a = b)
  |-------
{1}   pass(a!1) = Pas
{2}   seat_exists(aircraft(flt!1), seat(a!1))
```

Now we issue the ASSERT command:

```
Rule? (ASSERT)
Invoking decision procedures,
this simplifies to:
Cancel_assn_inv.1.1 :

[-1]   Flt = flt!1
```

```
[-2]   S1(Flt)(a!1)
[-3]   (FORALL (a: seat_assignment), (b: seat_assignment), (flt: flight):
         S1(flt)(a) AND S1(flt)(b) AND pass(a) = pass(b) IMPLIES a = b)
  |-------
[1]   pass(a!1) = Pas
[2]   seat_exists(aircraft(flt!1), seat(a!1))
{3}   TRUE

which is trivially true.

This completes the proof of Cancel_assn_inv.1.1.

Cancel_assn_inv.1.2 :

[-1]   (FORALL (a: seat_assignment), (flt: flight):
         S1(flt)(a) IMPLIES seat_exists(aircraft(flt), seat(a)))
[-2]   (FORALL (a: seat_assignment), (b: seat_assignment), (flt: flight):
         S1(flt)(a) AND S1(flt)(b) AND pass(a) = pass(b) IMPLIES a = b)
  |-------
{1}   Flt = flt!1
[2]   S1
        WITH [Flt :=
          {a: seat_assignment | S1(Flt)(a) AND pass(a) /= Pas}](flt!1)(a!1)
        IMPLIES seat_exists(aircraft(flt!1), seat(a!1))
```

PVS responds that Cancel_assn_inv.1.1 **is trivially true** and informs us that "**this completes the proof of** Cancel_assn_inv.1.1." However, our joy is shortlived because PVS quickly reminds us about Cancel_assn_inv.1.2. We are optimistic and try ASSERT:

```
Rule? (ASSERT)
Invoking decision procedures,
this simplifies to:
Cancel_assn_inv.1.2 :

[-1]   (FORALL (a: seat_assignment), (flt: flight):
         S1(flt)(a) IMPLIES seat_exists(aircraft(flt), seat(a)))
[-2]   (FORALL (a: seat_assignment), (b: seat_assignment), (flt: flight):
         S1(flt)(a) AND S1(flt)(b) AND pass(a) = pass(b) IMPLIES a = b)
  |-------
[1]   Flt = flt!1
{2}   S1(flt!1)(a!1) IMPLIES seat_exists(aircraft(flt!1), seat(a!1))
```

Well, it was worth a try. The ASSERT command at least simplified formula {2} considerably. We remember that the PVS decision procedures do not like universal quantifiers and proceed to eliminate them with a INST command:

```
Rule? (INST -1 "a!1" "flt!1")
Instantiating the top quantifier in -1 with the terms:
 (a!1 flt!1)
```

```
this simplifies to:
Cancel_assn_inv.1.2 :

{-1}  S1(flt!1)(a!1) IMPLIES seat_exists(aircraft(flt!1), seat(a!1))
[-2]   (FORALL (a: seat_assignment), (b: seat_assignment), (flt: flight):
         S1(flt)(a) AND S1(flt)(b) AND pass(a) = pass(b) IMPLIES a = b)
  |-------
[1]   Flt = flt!1
[2]   S1(flt!1)(a!1) IMPLIES seat_exists(aircraft(flt!1), seat(a!1))

which is trivially true.

This completes the proof of Cancel_assn_inv.1.2.


This completes the proof of Cancel_assn_inv.1.

Cancel_assn_inv.2 :

[-1]   (FORALL (a: seat_assignment), (flt: flight):
         S1(flt)(a) IMPLIES seat_exists(aircraft(flt), seat(a)))
[-2]   (FORALL (a: seat_assignment), (b: seat_assignment), (flt: flight):
         S1(flt)(a) AND S1(flt)(b) AND pass(a) = pass(b) IMPLIES a = b)
  |-------
{1}   (FORALL (a: seat_assignment), (b: seat_assignment), (flt: flight):
         S1
           WITH [Flt :=
             {a: seat_assignment | S1(Flt)(a) AND pass(a) /= Pas}](flt)(a)
           AND
           S1
             WITH [Flt :=
               {a: seat_assignment | S1(Flt)(a) AND pass(a) /= Pas}](flt)(b)
             AND pass(a) = pass(b)
           IMPLIES a = b)
```

PVS is satisfied that `Cancel_assn_inv.1.2` is true and supplies us with `Cancel_assn_inv.2` which is left from our earlier SPLIT command. As usual we begin by removing the quantifiers with a `SKOLEM` command:

```
Rule? (SKOLEM 1 ("AA" "B" "Flt2"))
For the top quantifier in 1, we introduce Skolem constants: (AA B Flt2)
this simplifies to:
Cancel_assn_inv.2 :

[-1]   (FORALL (a: seat_assignment), (flt: flight):
         S1(flt)(a) IMPLIES seat_exists(aircraft(flt), seat(a)))
[-2]   (FORALL (a: seat_assignment), (b: seat_assignment), (flt: flight):
         S1(flt)(a) AND S1(flt)(b) AND pass(a) = pass(b) IMPLIES a = b)
```

```
  |-------
{1}   S1
        WITH [Flt :=
          {a: seat_assignment | S1(Flt)(a) AND pass(a) /= Pas}](Flt2)(AA)
        AND
        S1
          WITH [Flt :=
            {a: seat_assignment | S1(Flt)(a) AND pass(a) /= Pas}](Flt2)(B)
          AND pass(AA) = pass(B)
        IMPLIES AA = B
```

We can see that formula {1} is closely related to [-2] but is complicated because of the function modifications (i.e the `WITH` clauses). So we decide to use the same strategy as before, case split on `Flt = Flt2`:

```
Rule? (CASE "Flt = Flt2")
Case splitting on
   Flt = Flt2,
this yields  2 subgoals:
Cancel_assn_inv.2.1 :

{-1}   Flt = Flt2
[-2]   (FORALL (a: seat_assignment), (flt: flight):
         S1(flt)(a) IMPLIES seat_exists(aircraft(flt), seat(a)))
[-3]   (FORALL (a: seat_assignment), (b: seat_assignment), (flt: flight):
         S1(flt)(a) AND S1(flt)(b) AND pass(a) = pass(b) IMPLIES a = b)
  |-------
[1]   S1
        WITH [Flt :=
          {a: seat_assignment | S1(Flt)(a) AND pass(a) /= Pas}](Flt2)(AA)
        AND
        S1
          WITH [Flt :=
            {a: seat_assignment | S1(Flt)(a) AND pass(a) /= Pas}](Flt2)(B)
          AND pass(AA) = pass(B)
        IMPLIES AA = B
```

We have two subgoals `Cancel_assn_inv.2.1` and `Cancel_assn_inv.2.2`. The system directs our attention to the `.1` formula. We issue an `ASSERT` command to collapse the `WITH` clauses in the presence of formula {-1}:

```
Rule? (ASSERT)
Invoking decision procedures,
this simplifies to:
Cancel_assn_inv.2.1 :

[-1]   Flt = Flt2
[-2]   (FORALL (a: seat_assignment), (flt: flight):
         S1(flt)(a) IMPLIES seat_exists(aircraft(flt), seat(a)))
```

```
[-3]   (FORALL (a: seat_assignment), (b: seat_assignment), (flt: flight):
          S1(flt)(a) AND S1(flt)(b) AND pass(a) = pass(b) IMPLIES a = b)
  |-------
{1}   (S1(Flt)(AA) AND pass(AA) /= Pas)
        AND (S1(Flt)(B) AND pass(B) /= Pas) AND pass(AA) = pass(B)
        IMPLIES AA = B
```

We need to get rid of the FORALL quantifier in formula [-3]. First, we must decide whether a skolemization or quantification is required. Here is the basic rule: FORALL quantifiers in formulas on the antecedent side and EXISTS quantifiers on the consequent side must be instantiated using INST (or the equivalent command (QUANT). EXISTS quantifiers in formulas on the antecedent side and FORALL quantifiers on the consequent side must be skolemized[10]. Thus, we need to use a INST command:

```
Rule? (INST -3 "AA" "B" "Flt")
Instantiating the top quantifier in -3 with the terms:
 (AA B Flt)
this simplifies to:
Cancel_assn_inv.2.1 :

[-1]   Flt = Flt2
[-2]   (FORALL (a: seat_assignment), (flt: flight):
          S1(flt)(a) IMPLIES seat_exists(aircraft(flt), seat(a)))
{-3}   S1(Flt)(AA) AND S1(Flt)(B) AND pass(AA) = pass(B) IMPLIES AA = B
  |-------
[1]   (S1(Flt)(AA) AND pass(AA) /= Pas)
        AND (S1(Flt)(B) AND pass(B) /= Pas) AND pass(AA) = pass(B)
        IMPLIES AA = B
```

We notice that there are several ANDs in the formula so we decide to flatten it before we ASSERT:

```
Rule? (FLATTEN)
Applying disjunctive simplification to flatten sequent,
this simplifies to:
Cancel_assn_inv.2.1 :

[-1]   Flt = Flt2
{-2}   S1(Flt)(AA)
{-3}   S1(Flt)(B)
{-4}   pass(AA) = pass(B)
[-5]   (FORALL (a: seat_assignment), (flt: flight):
          S1(flt)(a) IMPLIES seat_exists(aircraft(flt), seat(a)))
[-6]   S1(Flt)(AA) AND S1(Flt)(B) AND pass(AA) = pass(B) IMPLIES AA = B
  |-------
{1}   pass(AA) = Pas
{2}   pass(B) = Pas
{3}   AA = B
```

[10] If you guess wrong the theorem prover will promptly inform you.

We can see that {-2}, {-3} and {-4} will discharge the premise of [-6] yielding AA = B. This is identical to one of the consequent formulas, i.e.,{3}, so we should be done. We therefore issue an ASSERT command:

```
Rule? (ASSERT)
Invoking decision procedures,
this simplifies to:
Cancel_assn_inv.2.1 :

[-1]   Flt = Flt2
[-2]   S1(Flt)(AA)
[-3]   S1(Flt)(B)
[-4]   pass(AA) = pass(B)
[-5]   (FORALL (a: seat_assignment), (flt: flight):
         S1(flt)(a) IMPLIES seat_exists(aircraft(flt), seat(a)))
  |-------
[1]   pass(AA) = Pas
[2]   pass(B) = Pas
[3]   AA = B
{4}   TRUE

which is trivially true.

This completes the proof of Cancel_assn_inv.2.1.

Cancel_assn_inv.2.2 :

[-1]   (FORALL (a: seat_assignment), (flt: flight):
         S1(flt)(a) IMPLIES seat_exists(aircraft(flt), seat(a)))
[-2]   (FORALL (a: seat_assignment), (b: seat_assignment), (flt: flight):
         S1(flt)(a) AND S1(flt)(b) AND pass(a) = pass(b) IMPLIES a = b)
  |-------
{1}   Flt = Flt2
[2]   S1
        WITH [Flt :=
          {a: seat_assignment | S1(Flt)(a) AND pass(a) /= Pas}](Flt2)(AA)
        AND
        S1
          WITH [Flt :=
            {a: seat_assignment | S1(Flt)(a) AND pass(a) /= Pas}](Flt2)(B)
          AND pass(AA) = pass(B)
        IMPLIES AA = B
```

This finishes off Cancel_assn_inv.2.1 and we are onto Cancel_assn_inv.2.2. We decide to simplify with ASSERT:

```
Rule? (ASSERT)
Invoking decision procedures,
this simplifies to:
```

```
Cancel_assn_inv.2.2 :

[-1]   (FORALL (a: seat_assignment), (flt: flight):
         S1(flt)(a) IMPLIES seat_exists(aircraft(flt), seat(a)))
[-2]   (FORALL (a: seat_assignment), (b: seat_assignment), (flt: flight):
         S1(flt)(a) AND S1(flt)(b) AND pass(a) = pass(b) IMPLIES a = b)
  |-------
[1]    Flt = Flt2
{2}    S1(Flt2)(AA) AND S1(Flt2)(B) AND pass(AA) = pass(B) IMPLIES AA = B
```

Next, we must eliminate the quantifiers in [-2][11].

```
Rule? (INST -2 "AA" "B" "Flt2")
Instantiating the top quantifier in -2 with the terms:
 (AA B Flt2)
this simplifies to:
Cancel_assn_inv.2.2 :

[-1]   (FORALL (a: seat_assignment), (flt: flight):
         S1(flt)(a) IMPLIES seat_exists(aircraft(flt), seat(a)))
{-2}   S1(Flt2)(AA) AND S1(Flt2)(B) AND pass(AA) = pass(B) IMPLIES AA = B
  |-------
[1]    Flt = Flt2
[2]    S1(Flt2)(AA) AND S1(Flt2)(B) AND pass(AA) = pass(B) IMPLIES AA = B

which is trivially true.

This completes the proof of Cancel_assn_inv.2.2.


This completes the proof of Cancel_assn_inv.2.

Q.E.D.


Run time  = 25.77 secs.
Real time = 36.84 secs.

Cancel_assn_inv :

  |-------
{1}    (FORALL (flt: flight), (pas: passenger), (s1: assn_state):
          assn_invariant(s1) IMPLIES assn_invariant(Cancel_assn(flt, pas, s1)))
```

[11] The amount of typing required for this command can be reduced through use of the M-s command, which retrieves the previous commands. By issuing M-s M-s M-s M-s the system retrieves the command that was issued four times ago, i.e. (INST -3 "AA" "B" "Flt2"). This is easily changed into (INST -2 "AA" "B" "Flt2").

With the appearance of "Q.E.D." we know we have succeeded. Using the PVS command **M-x edit-proof** we can see the total structure of the proof. PVS displays the completed proof as follows:

```
("" (SKOLEM * ("Flt" "Pas" "S1"))
    (EXPAND "assn_invariant")
    (EXPAND "existence")
    (EXPAND "uniqueness")
    (EXPAND "Cancel_assn")
    (EXPAND "member")
    (FLATTEN)
    (SPLIT 1)
    (("1" (SKOLEM 1 ("a!1" "flt!1"))
          (CASE "Flt = flt!1")
          (("1" (ASSERT)
                (INST -2 "a!1" "Flt")
                (FLATTEN)
                (ASSERT)
                (PROPAX))
           ("2" (ASSERT)
                (INST -1 "a!1" "flt!1")
                (PROPAX))))
     ("2" (SKOLEM 1 ("AA" "B" "Flt2"))
          (CASE "Flt = Flt2")
          (("1" (ASSERT)
                (INST -3 "AA" "B" "Flt")
                (FLATTEN)
                (ASSERT)
                (PROPAX))
           ("2" (ASSERT)
                (INST -2 "AA" "B" "Flt2")
                (PROPAX))))))
```

This may be edited and rerun using the C-c C-c command.

3.2 Proof that Make_assn Maintains the Invariant

In this subsection we will prove the **Make_assn_inv** invariant:

```
Make_assn_inv: THEOREM assn_invariant(s1) =>
                         assn_invariant(Make_assn(flt,pas,pref,s1))
```

However, we will perform the proof in a slightly different manner this time—we will prove two lemmas before we attack the theorem. We are doing this because we have noticed that **assn_invariant** consists of two separate properties, **existence** and **uniqueness**:

```
assn_invariant: function[assn_state -> bool] =
 (LAMBDA as: existence(as) AND uniqueness(as))
```

that can be proved separately as lemmas:

```
MAe: THEOREM existence(s1)
            IMPLIES existence(Make_assn(flt,pas,pref,s1))

MAu: THEOREM uniqueness(s1)
            IMPLIES uniqueness(Make_assn(flt,pas,pref,s1))
```

Then, we will prove Make_assn_inv from these. The order of the proofs is not critical. However, many times it is valuable to prove that the main theorem follows from the lemmas so that one does not prove a useless lemma.

3.2.1 Proof of MAe

We begin with MAe

```
MAe :

  |-------
{1}   (FORALL (flt: flight), (pas: passenger),
              (pref: preference), (s1: assn_state):
        existence(s1) IMPLIES existence(Make_assn(flt, pas, pref, s1)))
```

As in the previous proof, we need to eliminate the universal quantifier by skolemization. However, we will use the SKOSIMP command to do this.

```
Rule? (SKOSIMP)
For the top quantifier in 1, we introduce Skolem constants: (flt!1 pas!1
pref!1 s1!1) this simplifies to:
MAe :

  |-------
{1}   existence(s1!1) IMPLIES existence(Make_assn(flt!1, pas!1, pref!1, s1!1))

Applying disjunctive simplification to flatten sequent,
this simplifies to:
MAe :

{-1}   existence(s1!1)
  |-------
{1}   existence(Make_assn(flt!1, pas!1, pref!1, s1!1))
```

The SKOSIMP command is equivalent to a SKOLEM! command followed by a FLATTEN command. Note that the names for the skolem constants are selected by the prover automatically. Next, we expand the definition of existence:

```
Rule? (EXPAND "existence")
Expanding the definition of existence
this simplifies to:
MAe :

{-1}   (FORALL (a: seat_assignment), (flt: flight):
```

```
          member(a, s1!1(flt)) IMPLIES seat_exists(aircraft(flt), seat(a)))
  |-------
{1}   (FORALL (a: seat_assignment), (flt: flight):
          member(a, Make_assn(flt!1, pas!1, pref!1, s1!1)(flt))
            IMPLIES seat_exists(aircraft(flt), seat(a)))
```

We expand the definition of Make_assn:

```
Rule? (EXPAND "Make_assn")
Expanding the definition of Make_assn
this simplifies to:
MAe :

[-1]   (FORALL (a: seat_assignment), (flt: flight):
          member(a, s1!1(flt)) IMPLIES seat_exists(aircraft(flt), seat(a)))
  |-------
{1}   (FORALL (a: seat_assignment), (flt: flight):
          member(a,
                 IF pref_filled(s1!1, flt!1, pref!1)
                      OR pass_on_flight(pas!1, flt!1, s1!1) THEN s1!1
                 ELSE s1!1
                     WITH [flt!1 :=
                       add((# seat := Next_seat(s1!1, flt!1, pref!1),
                              pass := pas!1
                              #), s1!1(flt!1))]
                 ENDIF(flt))
            IMPLIES seat_exists(aircraft(flt), seat(a)))
```

In the previous theorem we had to expand member several times. So this time we decide to make this automatic through use of the AUTO-REWRITE command:

```
Rule? (AUTO-REWRITE "member")
Installing automatic rewrites:
  member,
this simplifies to:
MAe :

[-1]   (FORALL (a: seat_assignment), (flt: flight):
          member(a, s1!1(flt)) IMPLIES seat_exists(aircraft(flt), seat(a)))
  |-------
[1]   (FORALL (a: seat_assignment), (flt: flight):
          member(a,
                 IF pref_filled(s1!1, flt!1, pref!1)
                      OR pass_on_flight(pas!1, flt!1, s1!1) THEN s1!1
                 ELSE s1!1
                     WITH [flt!1 :=
                       add((# seat := Next_seat(s1!1, flt!1, pref!1),
                              pass := pas!1
                              #), s1!1(flt!1))]
```

```
                ENDIF(flt))
            IMPLIES seat_exists(aircraft(flt), seat(a)))
```

Notice that AUTO-REWRITE does not immediately replace member with its definition. The rewrite will take place when one issues an ASSERT command. We issue another SKOSIMP command to eliminate the universal quantifiers in formula [1]:

```
Rule? (SKOSIMP)
For the top quantifier in 1, we introduce Skolem constants: (a!1 flt!2)
this simplifies to:
MAe :

[-1]   (FORALL (a: seat_assignment), (flt: flight):
         member(a, s1!1(flt)) IMPLIES seat_exists(aircraft(flt), seat(a)))
  |-------
{1}   member(a!1,
             IF pref_filled(s1!1, flt!1, pref!1)
                  OR pass_on_flight(pas!1, flt!1, s1!1) THEN s1!1
             ELSE s1!1
                 WITH [flt!1 :=
                   add((# seat := Next_seat(s1!1, flt!1, pref!1),
                          pass := pas!1
                          #), s1!1(flt!1))]
             ENDIF(flt!2))
        IMPLIES seat_exists(aircraft(flt!2), seat(a!1))

Applying disjunctive simplification to flatten sequent,
this simplifies to:
MAe :

[-1]   (FORALL (a: seat_assignment), (flt: flight):
         member(a, s1!1(flt)) IMPLIES seat_exists(aircraft(flt), seat(a)))
{-2}   member(a!1,
             IF pref_filled(s1!1, flt!1, pref!1)
                  OR pass_on_flight(pas!1, flt!1, s1!1) THEN s1!1
             ELSE s1!1
                 WITH [flt!1 :=
                   add((# seat := Next_seat(s1!1, flt!1, pref!1),
                          pass := pas!1
                          #), s1!1(flt!1))]
             ENDIF(flt!2))
  |-------
{1}   seat_exists(aircraft(flt!2), seat(a!1))
```

The universal quantifier in {-1} must be removed by quantification. We want the expression "seat_exists(aircraft(flt), seat(a)))" in formula [-1] to match formula {1}, so a!1 should be substituted for a and flt!2 for flt. For variety we will use the QUANT command rather than the INST command. Functionally they are identical; however, QUANT requires an extra layer of parentheses (i.e., (QUANT -1 ("a!1" "flt!2")) does the same thing as the (INST -1 "a!1" "flt!2") command).

```
Rule? (QUANT -1 ("a!1" "flt!2"))
Instantiating the top quantifier in -1 with the terms:
 (a!1 flt!2)
this simplifies to:
MAe :

{-1}   member(a!1, s1!1(flt!2)) IMPLIES seat_exists(aircraft(flt!2), seat(a!1))
[-2]   member(a!1,
             IF pref_filled(s1!1, flt!1, pref!1)
                  OR pass_on_flight(pas!1, flt!1, s1!1) THEN s1!1
             ELSE s1!1
                 WITH [flt!1 :=
                   add((# seat := Next_seat(s1!1, flt!1, pref!1),
                          pass := pas!1
                          #), s1!1(flt!1))]
             ENDIF(flt!2))
  |-------
[1]   seat_exists(aircraft(flt!2), seat(a!1))
```

We simplify with ASSERT:

```
Rule? (ASSERT)
Rewriting member(a!1, s1!1(flt!2)) to s1!1(flt!2)(a!1).
Rewriting member(a!1,
       IF pref_filled(s1!1, flt!1, pref!1)
             OR pass_on_flight(pas!1, flt!1, s1!1) THEN s1!1
       ELSE s1!1
           WITH [flt!1 :=
             add((# seat := Next_seat(s1!1, flt!1, pref!1), pass := pas!1
                   #), s1!1(flt!1))]
       ENDIF(flt!2)) to IF pref_filled(s1!1, flt!1, pref!1)
      OR pass_on_flight(pas!1, flt!1, s1!1) THEN s1!1
ELSE s1!1
    WITH [flt!1 :=
      add((# seat := Next_seat(s1!1, flt!1, pref!1), pass := pas!1
            #), s1!1(flt!1))]
ENDIF(flt!2)(a!1).
Invoking decision procedures,
this simplifies to:
MAe :

{-1}   IF pref_filled(s1!1, flt!1, pref!1)
            OR pass_on_flight(pas!1, flt!1, s1!1) THEN s1!1
      ELSE s1!1
          WITH [flt!1 :=
            add((# seat := Next_seat(s1!1, flt!1, pref!1), pass := pas!1
```

```
                    #), s1!1(flt!1))]
        ENDIF(flt!2)(a!1)
  |-------
{1}   s1!1(flt!2)(a!1)
[2]   seat_exists(aircraft(flt!2), seat(a!1))
```

We notice that the rewrites of member takes place at this time. We also notice that the IF THEN ELSE structure in formula {-1} is not at the outer most level (i.e. the text (flt!2)(a!1) follows the ENDIF), so we issue a LIFT-IF command:

```
Rule? (LIFT-IF -1)
Lifting IF-conditions to the top level,
this simplifies to:
MAe :

{-1}   IF pref_filled(s1!1, flt!1, pref!1)
            OR pass_on_flight(pas!1, flt!1, s1!1) THEN s1!1(flt!2)(a!1)
        ELSE
          s1!1
            WITH [flt!1 :=
              add((# seat := Next_seat(s1!1, flt!1, pref!1), pass := pas!1
                    #), s1!1(flt!1))](flt!2)(a!1)
        ENDIF
  |-------
[1]   s1!1(flt!2)(a!1)
[2]   seat_exists(aircraft(flt!2), seat(a!1))
```

Now that the IF THEN ELSE is at the outermost level it can be split into two sequents using the SPLIT command (i.e., IF A THEN B ELSE C ENDIF is equivalent to $A \supset B \wedge NOT\ A \supset C$.) Thus, we use a SPLIT command:

```
Rule? (SPLIT -1)
Splitting conjunctions,
this yields  2 subgoals:
MAe.1 :

{-1}   (pref_filled(s1!1, flt!1, pref!1) OR pass_on_flight(pas!1, flt!1, s1!1))
        AND s1!1(flt!2)(a!1)
  |-------
[1]   s1!1(flt!2)(a!1)
[2]   seat_exists(aircraft(flt!2), seat(a!1))
```

We issue the GROUND command to finish off the proof of this sequent:

```
Rule? (GROUND)
Applying propositional simplification and decision procedures,

This completes the proof of MAe.1.

MAe.2 :
```

```
{-1}   NOT
        (pref_filled(s1!1, flt!1, pref!1)
           OR pass_on_flight(pas!1, flt!1, s1!1))
        AND
        s1!1
          WITH [flt!1 :=
            add((# seat := Next_seat(s1!1, flt!1, pref!1), pass := pas!1
                   #), s1!1(flt!1))](flt!2)(a!1)
  |-------
[1]   s1!1(flt!2)(a!1)
[2]   seat_exists(aircraft(flt!2), seat(a!1))
```

The ground procedures simplify the sequent to the point where PVS recognizes the formula as true. PVS writes "`This completes the proof of MAe.1`" and turns our attention to `MAe.2`. Encouraged by our progress, we decide to expand `add` according to its definition:

```
Rule? (EXPAND "add")
Rewriting member(y, s1!1(flt!1)) to s1!1(flt!1)(y).
Expanding the definition of add
this simplifies to:
MAe.2 :

{-1}   NOT
        (pref_filled(s1!1, flt!1, pref!1)
           OR pass_on_flight(pas!1, flt!1, s1!1))
        AND
        s1!1
          WITH [flt!1 :=
            {y: [# seat: [row, position], pass: passenger #] |
            (# seat := Next_seat(s1!1, flt!1, pref!1), pass := pas!1 #) = y
              OR s1!1(flt!1)(y)}](flt!2)(a!1)
  |-------
[1]   s1!1(flt!2)(a!1)
[2]   seat_exists(aircraft(flt!2), seat(a!1))
```

There is nothing in the antecedent formula that will make [1] or [2] true by itself. Formula [2] asserts that the seat determined by `seat(a!1)` actually exists. But formula{-1} tells us that `seat(a!1)` is obtained from the `Next_seat` function. In our specification, we left these functions as uninterpreted functions. Earlier we stated that we would need a property about these functions in order to make the proofs go through. This is where we recognize this need. The desired property is also obvious—the property given in the `Next_seat_ax` axiom. We make this axiom available in the sequent by use of the `LEMMA` command:

```
Rule? (LEMMA "Next_seat_ax")
Applying Next_seat_ax where
this simplifies to:
MAe.2 :
```

```
{-1}   (FORALL (flt: flight), (pref: preference), (s1: assn_state):
         NOT pref_filled(s1, flt, pref)
           IMPLIES seat_exists(aircraft(flt), Next_seat(s1, flt, pref)))
[-2]   NOT
        (pref_filled(s1!1, flt!1, pref!1)
           OR pass_on_flight(pas!1, flt!1, s1!1))
        AND
        s1!1
          WITH [flt!1 :=
            {y: [# seat: [row, position], pass: passenger #] |
            (# seat := Next_seat(s1!1, flt!1, pref!1), pass := pas!1 #) = y
              OR s1!1(flt!1)(y)}](flt!2)(a!1)
  |-------
[1]    s1!1(flt!2)(a!1)
[2]    seat_exists(aircraft(flt!2), seat(a!1))
```

Whenever one introduces a lemma one usually must quantify the universal variables in this lemma:

```
Rule? (INST -1 "flt!1 " "pref!1" "s1!1")
Instantiating the top quantifier in -1 with the terms:
 (flt!1  pref!1 s1!1)
this simplifies to:
MAe.2 :

{-1}   NOT pref_filled(s1!1, flt!1, pref!1)
        IMPLIES seat_exists(aircraft(flt!1), Next_seat(s1!1, flt!1, pref!1))
[-2]   NOT
        (pref_filled(s1!1, flt!1, pref!1)
           OR pass_on_flight(pas!1, flt!1, s1!1))
        AND
        s1!1
          WITH [flt!1 :=
            {y: [# seat: [row, position], pass: passenger #] |
            (# seat := Next_seat(s1!1, flt!1, pref!1), pass := pas!1 #) = y
              OR s1!1(flt!1)(y)}](flt!2)(a!1)
  |-------
[1]    s1!1(flt!2)(a!1)
[2]    seat_exists(aircraft(flt!2), seat(a!1))
```

We now issue a GROUND command:

```
Rule? (GROUND)
Applying propositional simplification and decision procedures,
this simplifies to:
MAe.2 :

{-1}   seat_exists(aircraft(flt!1), Next_seat(s1!1, flt!1, pref!1))
{-2}   s1!1
        WITH [flt!1 :=
```

```
          {y: [# seat: [row, position], pass: passenger #] |
          (# seat := Next_seat(s1!1, flt!1, pref!1), pass := pas!1 #) = y
            OR s1!1(flt!1)(y)}](flt!2)(a!1)
  |-------
{1}   pref_filled(s1!1, flt!1, pref!1)
{2}   pass_on_flight(pas!1, flt!1, s1!1)
[3]   s1!1(flt!2)(a!1)
[4]   seat_exists(aircraft(flt!2), seat(a!1))
```

We notice that formula {-2} modifies s1!1 at flt!1 but then retrieves the value for flt!2. For all cases other than flt!1 = flt!2 this formula would be much simpler. Therefore, we perform a case split on flt!1 = flt!2:

```
Rule? (CASE "flt!1 = flt!2")
Case splitting on
   flt!1 = flt!2,
this yields  2 subgoals:
MAe.2.1 :

{-1}   flt!1 = flt!2
[-2]   seat_exists(aircraft(flt!1), Next_seat(s1!1, flt!1, pref!1))
[-3]   s1!1
        WITH [flt!1 :=
          {y: [# seat: [row, position], pass: passenger #] |
          (# seat := Next_seat(s1!1, flt!1, pref!1), pass := pas!1 #) = y
            OR s1!1(flt!1)(y)}](flt!2)(a!1)
  |-------
[1]   pref_filled(s1!1, flt!1, pref!1)
[2]   pass_on_flight(pas!1, flt!1, s1!1)
[3]   s1!1(flt!2)(a!1)
[4]   seat_exists(aircraft(flt!2), seat(a!1))
```

We issue a GROUND command to finish off this sequent:

```
Rule? (GROUND)
Applying propositional simplification and decision procedures,

This completes the proof of MAe.2.1.

MAe.2.2 :

[-1]   seat_exists(aircraft(flt!1), Next_seat(s1!1, flt!1, pref!1))
[-2]   s1!1
        WITH [flt!1 :=
          {y: [# seat: [row, position], pass: passenger #] |
          (# seat := Next_seat(s1!1, flt!1, pref!1), pass := pas!1 #) = y
            OR s1!1(flt!1)(y)}](flt!2)(a!1)
  |-------
{1}   flt!1 = flt!2
```

```
[2]    pref_filled(s1!1, flt!1, pref!1)
[3]    pass_on_flight(pas!1, flt!1, s1!1)
[4]    s1!1(flt!2)(a!1)
[5]    seat_exists(aircraft(flt!2), seat(a!1))
```

The theorem prover presents us with **MAe2.2**. We issue another **GROUND** command:

```
Rule? (GROUND)
Applying propositional simplification and decision procedures,

This completes the proof of MAe.2.2.


This completes the proof of MAe.2.

Q.E.D.


Run time  = 13.18 secs.
Real time = 23.47 secs.

MAe :

  |-------
{1}   (FORALL (flt: flight), (pas: passenger),
              (pref: preference), (s1: assn_state):
         existence(s1) IMPLIES existence(Make_assn(flt, pas, pref, s1)))
>
```

We are happy to see the arrival of "Q.E.D." but then remember that **MAu** and the main theorem still await us. The complete proof is displayed by **M-x edit-proof** as:

```
("" (SKOLEM 1
            ("flt!1" "pas!1" "pref!1" "s1!1"))
    (FLATTEN)
    (EXPAND "existence")
    (EXPAND "Make_assn")
    (AUTO-REWRITE "member")
    (SKOLEM 1 ("a!1" "flt!2"))
    (FLATTEN)
    (QUANT -1 ("a!1" "flt!2"))
    (ASSERT)
    (LIFT-IF -1)
    (SPLIT -1)
    (("1" (GROUND))
      ("2" (EXPAND "add")
           (LEMMA "Next_seat_ax")
           (INST -1 "flt!1 " "pref!1" "s1!1")
           (GROUND)
```

```
            (CASE "flt!1 = flt!2")
            (("1" (GROUND)) ("2" (GROUND))))))
```

3.2.2 Proof of MAu

This lemma is a little harder than MAe, but encouraged by past success we eagerly press on, issuing M-x pr on MAu:

```
MAu :

  |-------
{1}   (FORALL (flt: flight), (pas: passenger),
              (pref: preference), (s1: assn_state):
         uniqueness(s1) IMPLIES uniqueness(Make_assn(flt, pas, pref, s1)))
```

The first step is fairly routine by now—we eliminate the universal quantifiers:

```
Rule? (SKOSIMP)
For the top quantifier in 1, we introduce Skolem constants: (flt!1 pas!1
pref!1 s1!1) this simplifies to:
MAu :

  |-------
{1}   uniqueness(s1!1) IMPLIES uniqueness(Make_assn(flt!1, pas!1, pref!1, s1!1))

Applying disjunctive simplification to flatten sequent,
this simplifies to:
MAu :

{-1}   uniqueness(s1!1)
  |-------
{1}   uniqueness(Make_assn(flt!1, pas!1, pref!1, s1!1))
```

We know that the Make_assn function is defined using the sets theory in the prelude, so we decide to automate the expanding of the functions in this theory using the AUTO-REWRITE-THEORY command. This command is similar to the AUTO-REWRITE command except that instead of naming a particular function that is to be automatically expanded, one just provides the name of a theory. All of the functions in this theory will automatically be expanded when an ASSERT command is issued.

```
Rule? (AUTO-REWRITE-THEORY "sets[seat_assignment]")
Adding rewrites from theory sets[seat_assignment]
Adding rewrite rule member
Adding rewrite rule union
Adding rewrite rule intersection
Adding rewrite rule difference
Adding rewrite rule add
Adding rewrite rule remove
```

```
Adding rewrite rule singleton
Adding rewrite rule subset?
Adding rewrite rule strict_subset?
Adding rewrite rule empty?
Adding rewrite rule emptyset
Adding rewrite rule nonempty?
Adding rewrite rule fullset
Adding rewrite rule disjoint?
Adding rewrite rule extensionality
Auto-rewritten theory sets[seat_assignment]
Rewriting relative to the theories:
   sets[seat_assignment],
   NIL,
   NIL,
this simplifies to:
MAu :

[-1]   uniqueness(s1!1)
  |-------
[1]   uniqueness(Make_assn(flt!1, pas!1, pref!1, s1!1))
```

The prover lists the names of the functions from the **sets** theory that will automatically be expanded. We now expand **uniqueness**:

```
Rule? (EXPAND "uniqueness")
Rewriting member(a, s1!1(flt)) to s1!1(flt)(a).
Rewriting member(b, s1!1(flt)) to s1!1(flt)(b).
Rewriting member(a, Make_assn(flt!1, ...)(flt)) to Make_assn(flt!1, ...)(flt)(a).
Rewriting member(b, Make_assn(flt!1, ...)(flt)) to Make_assn(flt!1, ...)(flt)(b).
Expanding the definition of uniqueness
this simplifies to:
MAu :

{-1}   (FORALL (a: seat_assignment), (b: seat_assignment), (flt: flight):
         s1!1(flt)(a) AND s1!1(flt)(b) AND pass(a) = pass(b) IMPLIES a = b)
  |-------
{1}   (FORALL (a: seat_assignment), (b: seat_assignment), (flt: flight):
         Make_assn(flt!1, pas!1, pref!1, s1!1)(flt)(a)
           AND Make_assn(flt!1, pas!1, pref!1, s1!1)(flt)(b)
             AND pass(a) = pass(b)
           IMPLIES a = b)
```

We remove the universal quantifiers in formula {1} using **SKOSIMP**:

```
Rule? (SKOSIMP)
For the top quantifier in 1, we introduce Skolem constants: (a!1 b!1 flt!2)
this simplifies to:
MAu :
```

```
[-1]   (FORALL (a: seat_assignment), (b: seat_assignment), (flt: flight):
         s1!1(flt)(a) AND s1!1(flt)(b) AND pass(a) = pass(b) IMPLIES a = b)
  |-------
{1}   Make_assn(flt!1, pas!1, pref!1, s1!1)(flt!2)(a!1)
        AND Make_assn(flt!1, pas!1, pref!1, s1!1)(flt!2)(b!1)
          AND pass(a!1) = pass(b!1)
        IMPLIES a!1 = b!1

Applying disjunctive simplification to flatten sequent,
this simplifies to:
MAu :

[-1]   (FORALL (a: seat_assignment), (b: seat_assignment), (flt: flight):
         s1!1(flt)(a) AND s1!1(flt)(b) AND pass(a) = pass(b) IMPLIES a = b)
{-2}   Make_assn(flt!1, pas!1, pref!1, s1!1)(flt!2)(a!1)
{-3}   Make_assn(flt!1, pas!1, pref!1, s1!1)(flt!2)(b!1)
{-4}   pass(a!1) = pass(b!1)
  |-------
{1}   a!1 = b!1
```

We instantiate formula [-1] with the constants just created in the previous skolemization:

```
Rule? (INST -1 "a!1" "b!1" "flt!2")
Instantiating the top quantifier in -1 with the terms:
 (a!1 b!1 flt!2)
this simplifies to:
MAu :

{-1}   s1!1(flt!2)(a!1) AND s1!1(flt!2)(b!1) AND pass(a!1) = pass(b!1)
        IMPLIES a!1 = b!1
[-2]   Make_assn(flt!1, pas!1, pref!1, s1!1)(flt!2)(a!1)
[-3]   Make_assn(flt!1, pas!1, pref!1, s1!1)(flt!2)(b!1)
[-4]   pass(a!1) = pass(b!1)
  |-------
[1]   a!1 = b!1
```

We realize we aren't going much further until we expand **Make_assn**:

```
Rule? (EXPAND "Make_assn")
Expanding the definition of Make_assn
this simplifies to:
MAu :

[-1]   s1!1(flt!2)(a!1) AND s1!1(flt!2)(b!1) AND pass(a!1) = pass(b!1)
        IMPLIES a!1 = b!1
{-2}   IF pref_filled(s1!1, flt!1, pref!1)
            OR pass_on_flight(pas!1, flt!1, s1!1) THEN s1!1
      ELSE s1!1
          WITH [flt!1 :=
```

```
            add((# seat := Next_seat(s1!1, flt!1, pref!1), pass := pas!1
                   #), s1!1(flt!1))]
      ENDIF(flt!2)(a!1)
{-3}   IF pref_filled(s1!1, flt!1, pref!1)
            OR pass_on_flight(pas!1, flt!1, s1!1) THEN s1!1
      ELSE s1!1
          WITH [flt!1 :=
            add((# seat := Next_seat(s1!1, flt!1, pref!1), pass := pas!1
                   #), s1!1(flt!1))]
      ENDIF(flt!2)(b!1)
[-4]   pass(a!1) = pass(b!1)
  |-------
[1]    a!1 = b!1
```

We are ready for member to be rewritten so we issue an ASSERT command:

```
Rule? (ASSERT)
Invoking decision procedures,
this simplifies to:
MAu :

[-1]   IF pref_filled(s1!1, flt!1, pref!1)
            OR pass_on_flight(pas!1, flt!1, s1!1) THEN s1!1
      ELSE s1!1
          WITH [flt!1 :=
            add((# seat := Next_seat(s1!1, flt!1, pref!1), pass := pas!1
                   #), s1!1(flt!1))]
      ENDIF(flt!2)(a!1)
[-2]   IF pref_filled(s1!1, flt!1, pref!1)
            OR pass_on_flight(pas!1, flt!1, s1!1) THEN s1!1
      ELSE s1!1
          WITH [flt!1 :=
            add((# seat := Next_seat(s1!1, flt!1, pref!1), pass := pas!1
                   #), s1!1(flt!1))]
      ENDIF(flt!2)(b!1)
[-3]   pass(a!1) = pass(b!1)
  |-------
{1}    s1!1(flt!2)(a!1) AND s1!1(flt!2)(b!1)
[2]    a!1 = b!1
```

We notice that the IF-THEN-ELSE structures in formulas {-1} and {-2} are not at the outermost level, so we issue a LIFT-IF command:

```
Rule? (LIFT-IF -1 -2)
Lifting IF-conditions to the top level,
this simplifies to:
MAu :

{-1}   IF pref_filled(s1!1, flt!1, pref!1)
```

```
            OR pass_on_flight(pas!1, flt!1, s1!1) THEN s1!1(flt!2)(a!1)
      ELSE
        s1!1
          WITH [flt!1 :=
            add((# seat := Next_seat(s1!1, flt!1, pref!1), pass := pas!1
                  #), s1!1(flt!1))](flt!2)(a!1)
      ENDIF
{-2}   IF pref_filled(s1!1, flt!1, pref!1)
            OR pass_on_flight(pas!1, flt!1, s1!1) THEN s1!1(flt!2)(b!1)
      ELSE
        s1!1
          WITH [flt!1 :=
            add((# seat := Next_seat(s1!1, flt!1, pref!1), pass := pas!1
                  #), s1!1(flt!1))](flt!2)(b!1)
      ENDIF
[-3]   pass(a!1) = pass(b!1)
  |-------
[1]    s1!1(flt!2)(a!1) AND s1!1(flt!2)(b!1)
[2]    a!1 = b!1
```

A case split on `flt!1 = flt!2` seems in order:

```
Rule? (CASE "flt!1 = flt!2")
Case splitting on
   flt!1 = flt!2,
this yields  2 subgoals:
MAu.1 :

{-1}   flt!1 = flt!2
[-2]   IF pref_filled(s1!1, flt!1, pref!1)
            OR pass_on_flight(pas!1, flt!1, s1!1) THEN s1!1(flt!2)(a!1)
      ELSE
        s1!1
          WITH [flt!1 :=
            add((# seat := Next_seat(s1!1, flt!1, pref!1), pass := pas!1
                  #), s1!1(flt!1))](flt!2)(a!1)
      ENDIF
[-3]   IF pref_filled(s1!1, flt!1, pref!1)
            OR pass_on_flight(pas!1, flt!1, s1!1) THEN s1!1(flt!2)(b!1)
      ELSE
        s1!1
          WITH [flt!1 :=
            add((# seat := Next_seat(s1!1, flt!1, pref!1), pass := pas!1
                  #), s1!1(flt!1))](flt!2)(b!1)
      ENDIF
[-4]   pass(a!1) = pass(b!1)
  |-------
[1]    s1!1(flt!2)(a!1) AND s1!1(flt!2)(b!1)
```

```
[2]    a!1 = b!1
```

We now collapse the IF THEN ELSE structure with a GROUND command:

```
Rule? (ground)
Applying propositional simplification and decision procedures,
this yields  2 subgoals:
MAu.1.1 :

{-1}   s1!1
        WITH [flt!1 :=
          add((# seat := Next_seat(s1!1, flt!1, pref!1), pass := pas!1
                 #), s1!1(flt!1))](flt!2)(b!1)
{-2}   s1!1
        WITH [flt!1 :=
          add((# seat := Next_seat(s1!1, flt!1, pref!1), pass := pas!1
                 #), s1!1(flt!1))](flt!2)(a!1)
[-3]   flt!1 = flt!2
[-4]   pass(a!1) = pass(b!1)
  |-------
{1}   pref_filled(s1!1, flt!1, pref!1)
{2}   pass_on_flight(pas!1, flt!1, s1!1)
{3}   pref_filled(s1!1, flt!1, pref!1)
{4}   pass_on_flight(pas!1, flt!1, s1!1)
{5}   s1!1(flt!2)(a!1)
[6]   a!1 = b!1
```

The GROUND command produces two subgoals. We notice that formulas {3} and {4 } are identical to formulas {1} and {2}, so we hide {3} and {4} to remove the clutter:

```
Rule? (HIDE 3 4)
Hiding formulas:  3, 4,
this simplifies to:
MAu.1.1 :

[-1]   s1!1
        WITH [flt!1 :=
          add((# seat := Next_seat(s1!1, flt!1, pref!1), pass := pas!1
                 #), s1!1(flt!1))](flt!2)(b!1)
[-2]   s1!1
        WITH [flt!1 :=
          add((# seat := Next_seat(s1!1, flt!1, pref!1), pass := pas!1
                 #), s1!1(flt!1))](flt!2)(a!1)
[-3]   flt!1 = flt!2
[-4]   pass(a!1) = pass(b!1)
  |-------
[1]   pref_filled(s1!1, flt!1, pref!1)
[2]   pass_on_flight(pas!1, flt!1, s1!1)
[3]   s1!1(flt!2)(a!1)
[4]   a!1 = b!1
```

We need to establish that `a!1 = b!1` given that `pass(a!1) = pass(b!1)`. We remember that `pass_on_flight` constrains the passenger fields to be unique, so we expand it:

```
Rule? (EXPAND "pass_on_flight")
Rewriting member(a, s1!1(flt!1)) to s1!1(flt!1)(a).
Expanding the definition of pass_on_flight
this simplifies to:
MAu.1.1 :

[-1]   s1!1
        WITH [flt!1 :=
          add((# seat := Next_seat(s1!1, flt!1, pref!1), pass := pas!1
                 #), s1!1(flt!1))](flt!2)(b!1)
[-2]   s1!1
        WITH [flt!1 :=
          add((# seat := Next_seat(s1!1, flt!1, pref!1), pass := pas!1
                 #), s1!1(flt!1))](flt!2)(a!1)
[-3]   flt!1 = flt!2
[-4]   pass(a!1) = pass(b!1)
  |-------
[1]   pref_filled(s1!1, flt!1, pref!1)
{2}   (EXISTS (a: seat_assignment): pass(a) = pas!1 AND s1!1(flt!1)(a))
[3]   s1!1(flt!2)(a!1)
[4]   a!1 = b!1
```

We are ready to instantiate formula {2}, but then realize that we are going to need two instances of it, one for `a!1` and one for `b!1`[12]. Thus, we will use the `INST-CP` command which saves the original form of the formula in addition to the instantiated form:

```
Rule? (INST-CP 2 "a!1")
Instantiating the top quantifier in 2 with the terms:
 a!1, ,
this simplifies to:
MAu.1.1 :

[-1]   s1!1
        WITH [flt!1 :=
          add((# seat := Next_seat(s1!1, flt!1, pref!1), pass := pas!1
                 #), s1!1(flt!1))](flt!2)(b!1)
[-2]   s1!1
        WITH [flt!1 :=
          add((# seat := Next_seat(s1!1, flt!1, pref!1), pass := pas!1
                 #), s1!1(flt!1))](flt!2)(a!1)
[-3]   flt!1 = flt!2
[-4]   pass(a!1) = pass(b!1)
  |-------
[1]   pref_filled(s1!1, flt!1, pref!1)
```

[12] It actually took me about an hour to figure this out.

```
[2]    (EXISTS (a: seat_assignment): pass(a) = pas!1 AND s1!1(flt!1)(a))
{3}    pass(a!1) = pas!1 AND s1!1(flt!1)(a!1)
[4]    s1!1(flt!2)(a!1)
[5]    a!1 = b!1
```

Now we can instantiate it with b!1 as well:

```
Rule? (INST 2 "b!1")
Instantiating the top quantifier in 2 with the terms:
 b!1,
,
this simplifies to:
MAu.1.1 :

[-1]    s1!1
         WITH [flt!1 :=
           add((# seat := Next_seat(s1!1, flt!1, pref!1), pass := pas!1
                  #), s1!1(flt!1))](flt!2)(b!1)
[-2]    s1!1
         WITH [flt!1 :=
           add((# seat := Next_seat(s1!1, flt!1, pref!1), pass := pas!1
                  #), s1!1(flt!1))](flt!2)(a!1)
[-3]    flt!1 = flt!2
[-4]    pass(a!1) = pass(b!1)
  |-------
[1]    pref_filled(s1!1, flt!1, pref!1)
{2}    pass(b!1) = pas!1 AND s1!1(flt!1)(b!1)
[3]    pass(a!1) = pas!1 AND s1!1(flt!1)(a!1)
[4]    s1!1(flt!2)(a!1)
[5]    a!1 = b!1
```

We issue an ASSERT command to complete the proof of this sequent:

```
Rule? (ASSERT)
Rewriting member(b!1, s1!1(flt!1)) to s1!1(flt!1)(b!1).
Rewriting add((# seat := Next_seat(s1!1, flt!1, pref!1), pass := pas!1 #),
  s1!1(flt!1))(b!1) to (# seat := Next_seat(s1!1, flt!1, pref!1),
    pass := pas!1 #) = b!1 OR s1!1(flt!1)(b!1).
Rewriting member(a!1, s1!1(flt!1)) to FALSE.
Rewriting add((# seat := Next_seat(s1!1, flt!1, pref!1), pass := pas!1 #),
  s1!1(flt!1))(a!1) to (# seat := Next_seat(s1!1, flt!1, pref!1),
    pass := pas!1 #) = a!1.
Invoking decision procedures,

This completes the proof of MAu.1.1.

MAu.1.2 :
```

```
{-1}   s1!1
        WITH [flt!1 :=
          add((# seat := Next_seat(s1!1, flt!1, pref!1), pass := pas!1
                 #), s1!1(flt!1))](flt!2)(b!1)
{-2}   s1!1
        WITH [flt!1 :=
          add((# seat := Next_seat(s1!1, flt!1, pref!1), pass := pas!1
                 #), s1!1(flt!1))](flt!2)(a!1)
[-3]   flt!1 = flt!2
[-4]   pass(a!1) = pass(b!1)
  |-------
{1}   pref_filled(s1!1, flt!1, pref!1)
{2}   pass_on_flight(pas!1, flt!1, s1!1)
{3}   pref_filled(s1!1, flt!1, pref!1)
{4}   pass_on_flight(pas!1, flt!1, s1!1)
{5}   s1!1(flt!2)(b!1)
[6]   a!1 = b!1
```

PVS tells us, "`This completes the proof of MAu.1.1`" and presents `MAu.1.2` to us. Once again we notice that formulas {3} and {4 } are identical to formulas {1} and {2}, so we hide {3} and {4} to remove the clutter:

```
Rule? (HIDE 3 4)
Hiding formulas:  3, 4,
this simplifies to:
MAu.1.2 :

[-1]   s1!1
        WITH [flt!1 :=
          add((# seat := Next_seat(s1!1, flt!1, pref!1), pass := pas!1
                 #), s1!1(flt!1))](flt!2)(b!1)
[-2]   s1!1
        WITH [flt!1 :=
          add((# seat := Next_seat(s1!1, flt!1, pref!1), pass := pas!1
                 #), s1!1(flt!1))](flt!2)(a!1)
[-3]   flt!1 = flt!2
[-4]   pass(a!1) = pass(b!1)
  |-------
[1]   pref_filled(s1!1, flt!1, pref!1)
[2]   pass_on_flight(pas!1, flt!1, s1!1)
[3]   s1!1(flt!2)(b!1)
[4]   a!1 = b!1
```

As in the other subgoal, we expand `pass_on_flight`:

```
Rule? (EXPAND "pass_on_flight")
Rewriting member(a, s1!1(flt!1)) to s1!1(flt!1)(a).
Expanding the definition of pass_on_flight
this simplifies to:
```

```
MAu.1.2 :

[-1]   s1!1
        WITH [flt!1 :=
          add((# seat := Next_seat(s1!1, flt!1, pref!1), pass := pas!1
                 #), s1!1(flt!1))](flt!2)(b!1)
[-2]   s1!1
        WITH [flt!1 :=
          add((# seat := Next_seat(s1!1, flt!1, pref!1), pass := pas!1
                 #), s1!1(flt!1))](flt!2)(a!1)
[-3]   flt!1 = flt!2
[-4]   pass(a!1) = pass(b!1)
  |-------
[1]   pref_filled(s1!1, flt!1, pref!1)
{2}   (EXISTS (a: seat_assignment): pass(a) = pas!1 AND s1!1(flt!1)(a))
[3]   s1!1(flt!2)(b!1)
[4]   a!1 = b!1
```

and doubly instantiate formula 2:

```
Rule? (INST-CP 2 "a!1")
Instantiating the top quantifier in 2 with the terms:
 a!1, ,
this simplifies to:
MAu.1.2 :

[-1]   s1!1
        WITH [flt!1 :=
          add((# seat := Next_seat(s1!1, flt!1, pref!1), pass := pas!1
                 #), s1!1(flt!1))](flt!2)(b!1)
[-2]   s1!1
        WITH [flt!1 :=
          add((# seat := Next_seat(s1!1, flt!1, pref!1), pass := pas!1
                 #), s1!1(flt!1))](flt!2)(a!1)
[-3]   flt!1 = flt!2
[-4]   pass(a!1) = pass(b!1)
  |-------
[1]   pref_filled(s1!1, flt!1, pref!1)
[2]   (EXISTS (a: seat_assignment): pass(a) = pas!1 AND s1!1(flt!1)(a))
{3}   pass(a!1) = pas!1 AND s1!1(flt!1)(a!1)
[4]   s1!1(flt!2)(b!1)
[5]   a!1 = b!1

Rule? (INST 2 "b!1")
Instantiating the top quantifier in 2 with the terms:
 b!1,
,
this simplifies to:
```

```
MAu.1.2 :

[-1]   s1!1
        WITH [flt!1 :=
          add((# seat := Next_seat(s1!1, flt!1, pref!1), pass := pas!1
                 #), s1!1(flt!1))](flt!2)(b!1)
[-2]   s1!1
        WITH [flt!1 :=
          add((# seat := Next_seat(s1!1, flt!1, pref!1), pass := pas!1
                 #), s1!1(flt!1))](flt!2)(a!1)
[-3]   flt!1 = flt!2
[-4]   pass(a!1) = pass(b!1)
  |-------
[1]   pref_filled(s1!1, flt!1, pref!1)
{2}   pass(b!1) = pas!1 AND s1!1(flt!1)(b!1)
[3]   pass(a!1) = pas!1 AND s1!1(flt!1)(a!1)
[4]   s1!1(flt!2)(b!1)
[5]   a!1 = b!1
```

We issue an ASSERT command to simplify:

```
Rule? (ASSERT)
Rewriting member(b!1, s1!1(flt!1)) to FALSE.
Rewriting add((# seat := Next_seat(s1!1, flt!1, pref!1), pass := pas!1 #),
  s1!1(flt!1))(b!1) to (# seat := Next_seat(s1!1, flt!1, pref!1),
    pass := pas!1 #) = b!1.
Rewriting member(a!1, s1!1(flt!1)) to s1!1(flt!1)(a!1).
Rewriting add((# seat := Next_seat(s1!1, flt!1, pref!1), pass := pas!1 #),
  s1!1(flt!1))(a!1) to s1!1(flt!1)(a!1).
Invoking decision procedures,
this simplifies to:
MAu.1.2 :

{-1}   (# seat := Next_seat(s1!1, flt!1, pref!1), pass := pas!1 #) = b!1
{-2}   s1!1(flt!1)(a!1)
[-3]   flt!1 = flt!2
[-4]   pass(a!1) = pass(b!1)
  |-------
[1]   pref_filled(s1!1, flt!1, pref!1)
{2}   TRUE
[3]   s1!1(flt!2)(b!1)
[4]   a!1 = b!1

which is trivially true.

This completes the proof of MAu.1.2.
```

```
This completes the proof of MAu.1.

MAu.2 :

[-1]   IF pref_filled(s1!1, flt!1, pref!1)
            OR pass_on_flight(pas!1, flt!1, s1!1) THEN s1!1(flt!2)(a!1)
      ELSE
        s1!1
          WITH [flt!1 :=
            add((# seat := Next_seat(s1!1, flt!1, pref!1), pass := pas!1
                   #), s1!1(flt!1))](flt!2)(a!1)
      ENDIF
[-2]   IF pref_filled(s1!1, flt!1, pref!1)
            OR pass_on_flight(pas!1, flt!1, s1!1) THEN s1!1(flt!2)(b!1)
      ELSE
        s1!1
          WITH [flt!1 :=
            add((# seat := Next_seat(s1!1, flt!1, pref!1), pass := pas!1
                   #), s1!1(flt!1))](flt!2)(b!1)
      ENDIF
[-3]   pass(a!1) = pass(b!1)
  |-------
{1}   flt!1 = flt!2
[2]   s1!1(flt!2)(a!1) AND s1!1(flt!2)(b!1)
[3]   a!1 = b!1
```

We are rewarded by our perspicaciousness. PVS informs us that the proof of MAu.1.2 is complete and thus MAu.1. as well. It then presents us with MAu.2. We issue an ASSERT command to simplify:

```
Rule?  (ASSERT)
Invoking decision procedures,
this simplifies to:
MAu.2 :

{-1}   s1!1(flt!2)(a!1)
{-2}   s1!1(flt!2)(b!1)
[-3]   pass(a!1) = pass(b!1)
  |-------
[1]   flt!1 = flt!2
[2]   s1!1(flt!2)(a!1) AND s1!1(flt!2)(b!1)
[3]   a!1 = b!1
```

We issue another ASSERT command:

```
Rule? (ASSERT)
Invoking decision procedures,
this simplifies to:
MAu.2 :
```

```
[-1]   s1!1(flt!2)(a!1)
[-2]   s1!1(flt!2)(b!1)
[-3]   pass(a!1) = pass(b!1)
  |-------
[1]   flt!1 = flt!2
{2}   TRUE
[3]   a!1 = b!1

which is trivially true.

This completes the proof of MAu.2.

Q.E.D.


Run time  = 45.09 secs.
Real time = 291.16 secs.

MAu :

  |-------
{1}   (FORALL (flt: flight), (pas: passenger),
              (pref: preference), (s1: assn_state):
         uniqueness(s1) IMPLIES uniqueness(Make_assn(flt, pas, pref, s1)))
>
```

M-x edit-pr displays the complete proof as follows:

```
("" (SKOLEM 1
           ("flt!1" "pas!1" "pref!1" "s1!1"))
    (FLATTEN)
    (AUTO-REWRITE-THEORY "sets[seat_assignment]")
    (EXPAND "uniqueness")
    (SKOLEM 1 ("a!1" "b!1" "flt!2"))
    (FLATTEN)
    (INST -1 "a!1" "b!1" "flt!2")
    (EXPAND "Make_assn")
    (ASSERT)
    (LIFT-IF -1 -2)
    (CASE "flt!1 = flt!2")
    (("1" (GROUND)
          (("1" (HIDE 3 4)
                (EXPAND "pass_on_flight")
                (INST-CP 2 "a!1")
                (INST 2 "b!1")
                (ASSERT))
            ("2" (HIDE 3 4)
                 (EXPAND "pass_on_flight")
```

```
                (INST-CP 2 "a!1")
                (INST 2 "b!1")
                (ASSERT)
                (PROPAX))))
     ("2" (ASSERT) (ASSERT) (PROPAX))))
```

This completes the two lemmas.

3.2.3 Proof of Theorem

We now must show that these two lemmas imply **Make_assn_inv**. This is very straight forward and the details are left to the reader. Hint: **M-x edit-pr** on this theorem gives

```
("" (SKOSIMP)
    (EXPAND "assn_invariant")
    (LEMMA "MAe")
    (INST -1 "flt!1 " "pas!1 " "pref!1 " "s1!1")
    (LEMMA "MAu")
    (INST -1 "flt!1 " "pas!1 " "pref!1 " "s1!1")
    (GROUND))
```

3.3 Proof that the Initial State Satisfies the Invariant

It is necessary to show that the initial state of the system satisfies the invariant. This together with the invariant-preserving properties about the operations are sufficient to establish that the system will *always* preserve the invariant. The needed theorem for the initial state is:

```
initial_state_inv: THEOREM assn_invariant(initial_state)
```

This theorem is easy to prove. In fact, the PVS strategy for proving TCCs almost proves it without help. This strategy is automatically invoked when one issues a **M-x tcp** command to prove the TCCs. However, this strategy is also available during interactive proof by typing **(TCC)**:

```
initial_state_inv :

  |-------
{1}   assn_invariant(initial_state)

Rule? (TCC)
Rewriting initial_state(flt) to emptyset[seat_assignment].
Rewriting emptyset[seat_assignment](a) to FALSE.
Rewriting member(a, emptyset[seat_assignment]) to FALSE.
Rewriting existence(initial_state) to
  (FORALL (a: seat_assignment), (flt: flight): TRUE).
Rewriting initial_state(flt) to emptyset[seat_assignment].
Rewriting emptyset[seat_assignment](a) to FALSE.
Rewriting member(a, emptyset[seat_assignment]) to FALSE.
Rewriting initial_state(flt) to emptyset[seat_assignment].
Rewriting emptyset[seat_assignment](b) to FALSE.
Rewriting member(b, emptyset[seat_assignment]) to FALSE.
```

```
Rewriting uniqueness(initial_state) to
  (FORALL (a: seat_assignment), (b: seat_assignment), (flt: flight): TRUE).
Rewriting assn_invariant(initial_state) to
  (FORALL (a: seat_assignment), (flt: flight): TRUE)
  AND (FORALL (a: seat_assignment), (b: seat_assignment), (flt: flight): TRUE).

Trying repeated skolemization, instantiation, and if-lifting,
this yields  2 subgoals:
initial_state_inv.1 :

  |-------
{1}   (FORALL (a: seat_assignment), (flt: flight): TRUE)
```

The TCC strategy has created two subgoals. Each of these are completed by issuing a single SKOSIMP command:

```
Rule? (SKOSIMP)
For the top quantifier in 1, we introduce Skolem constants: (a!1 flt!1)
this simplifies to:
initial_state_inv.1 :

  |-------
{1}   TRUE

which is trivially true.

This completes the proof of initial_state_inv.1.

initial_state_inv.2 :

  |-------
{1}   (FORALL (a: seat_assignment), (b: seat_assignment), (flt: flight): TRUE)

Rule? (SKOSIMP)
For the top quantifier in 1, we introduce Skolem constants: (a!1 b!1 flt!1)
this simplifies to:
initial_state_inv.2 :

  |-------
{1}   TRUE

which is trivially true.

This completes the proof of initial_state_inv.2.

Q.E.D.
```

```
Run time  = 3.78 secs.
Real time = 5.74 secs.
NIL
>
```

3.4 Proof of one_per_seat Invariant

The third invariant of the system has not been dealt with in the previous sections. As mentioned earlier, the proofs of these theorems are left to the reader for an exercise. The required theorems for the **Cancel_assn** and **Make_assn** operations are

```
Cancel_inv_one_per_seat: THEOREM one_per_seat(s1)
  IMPLIES one_per_seat(Cancel_assn(flt,pas,s1))


Make_inv_one_per_seat: THEOREM one_per_seat(s1)
IMPLIES one_per_seat(Make_assn(flt,pas,pref,s1))

initial_one_per_seat: THEOREM one_per_seat(initial_state)
```

An additional property about the uninterpreted function **Next_seat** must be added to the specification in order to prove **Make_inv_one_per_seat**:

```
Next_seat_ax_2: AXIOM (FORALL a: member(a,s1(flt)) IMPLIES
                                  seat(a) /= Next_seat(s1,flt,pref))
```

3.5 System Properties and Putative Theorems

Usually there are several types of system properties that are of interest to formalize and prove:

1. Properties about critical system operation derived from high level requirements
2. *Putative theorems* used to confirm our understanding of the specified system

An example of (2) is the property that if the system is in state **s1**, and we make a seat assignment and then immediately cancel it, we should return to the same system state:

```
Make_Cancel: THEOREM NOT pass_on_flight(pas,flt,s1) =>
                 Cancel_assn(flt,pas,Make_assn(flt,pas,pref,s1)) = s1
```

The proof of this theorem involves several new concepts not encountered in the previous proofs. The novice reader is encouraged to continue working at the terminal, while reading the following proof. A key difference in this proof is the need to establish the equality of functions. This requires the use of "extensionality" axioms provided by PVS. We issue the **M-x pr** command:

```
Make_Cancel :

  |-------
{1}   (FORALL (flt: flight), (pas: passenger),
              (pref: preference), (s1: assn_state):
          NOT pass_on_flight(pas, flt, s1)
            => Cancel_assn(flt, pas, Make_assn(flt, pas, pref, s1)) = s1)
```

As always we skolemize with SKOSIMP:

```
Rule?: (SKOSIMP)
For the top quantifier in 1, we introduce Skolem constants: (flt!1 pas!1 pref!1 s1!1)
this simplifies to:
Make_Cancel :

  |-------
{1}   NOT pass_on_flight(pas!1, flt!1, s1!1)
        => Cancel_assn(flt!1, pas!1, Make_assn(flt!1, pas!1, pref!1, s1!1))
          = s1!1

Applying disjunctive simplification to flatten sequent,
this simplifies to:
Make_Cancel :

  |-------
{1}   pass_on_flight(pas!1, flt!1, s1!1)
{2}   Cancel_assn(flt!1, pas!1, Make_assn(flt!1, pas!1, pref!1, s1!1)) = s1!1
```

We expand Cancel_assn, pass_on_flight and Make_assn:

```
Rule?: (EXPAND "Cancel_assn")
Expanding the definition of Cancel_assn
this simplifies to:
Make_Cancel :

  |-------
[1]   pass_on_flight(pas!1, flt!1, s1!1)
{2}   Make_assn(flt!1, pas!1, pref!1, s1!1)
        WITH [flt!1 :=
          {a: seat_assignment |
          member(a, Make_assn(flt!1, pas!1, pref!1, s1!1)(flt!1))
            AND pass(a) /= pas!1}]
        = s1!1

Rule?: (EXPAND "pass_on_flight")
Expanding the definition of pass_on_flight
this simplifies to:
Make_Cancel :

  |-------
{1}   (EXISTS (a: seat_assignment): pass(a) = pas!1 AND member(a, s1!1(flt!1)))
[2]   Make_assn(flt!1, pas!1, pref!1, s1!1)
        WITH [flt!1 :=
          {a: seat_assignment |
          member(a, Make_assn(flt!1, pas!1, pref!1, s1!1)(flt!1))
```

```
            AND pass(a) /= pas!1}]
        = s1!1

Rule?: (EXPAND "Make_assn")
Expanding the definition of Make_assn
this simplifies to:
Make_Cancel :

  |-------
[1]   (EXISTS (a: seat_assignment): pass(a) = pas!1 AND member(a, s1!1(flt!1)))
{2}   IF pref_filled(s1!1, flt!1, pref!1)
            OR pass_on_flight(pas!1, flt!1, s1!1) THEN s1!1
      ELSE s1!1
          WITH [flt!1 :=
            add((# seat := Next_seat(s1!1, flt!1, pref!1), pass := pas!1
                   #), s1!1(flt!1))]
      ENDIF
        WITH [flt!1 :=
          {a: seat_assignment |
          member(a,
                 IF pref_filled(s1!1, flt!1, pref!1)
                       OR pass_on_flight(pas!1, flt!1, s1!1) THEN s1!1
                 ELSE s1!1
                     WITH [flt!1 :=
                       add((# seat := Next_seat(s1!1, flt!1, pref!1),
                              pass := pas!1
                              #), s1!1(flt!1))]
                 ENDIF(flt!1))
            AND pass(a) /= pas!1}]
        = s1!1
```

We issue an (AUTO-REWRITE-THEORY "sets[seat_assignment]") directive to the prover:

```
Rule?: (AUTO-REWRITE-THEORY "sets[seat_assignment]")
Adding rewrites from theory sets[seat_assignment]
Adding rewrite rule member
Adding rewrite rule union
Adding rewrite rule intersection
Adding rewrite rule difference
Adding rewrite rule add
Adding rewrite rule remove
Adding rewrite rule singleton
Adding rewrite rule subset?
Adding rewrite rule strict_subset?
Adding rewrite rule empty?
Adding rewrite rule emptyset
Adding rewrite rule nonempty?
Adding rewrite rule fullset
```

```
Adding rewrite rule disjoint?
Adding rewrite rule extensionality
Auto-rewritten theory sets[seat_assignment]
Rewriting relative to the theories:
   sets[seat_assignment],
   NIL,
   NIL,
this simplifies to:
Make_Cancel :

  |-------
[1]    (EXISTS (a: seat_assignment): pass(a) = pas!1 AND member(a, s1!1(flt!1)))
[2]    IF pref_filled(s1!1, flt!1, pref!1)
             OR pass_on_flight(pas!1, flt!1, s1!1) THEN s1!1
       ELSE s1!1
           WITH [flt!1 :=
             add((# seat := Next_seat(s1!1, flt!1, pref!1), pass := pas!1
                    #), s1!1(flt!1))]
       ENDIF
         WITH [flt!1 :=
           {a: seat_assignment |
           member(a,
                  IF pref_filled(s1!1, flt!1, pref!1)
                        OR pass_on_flight(pas!1, flt!1, s1!1) THEN s1!1
                  ELSE s1!1
                      WITH [flt!1 :=
                        add((# seat := Next_seat(s1!1, flt!1, pref!1),
                               pass := pas!1
                               #), s1!1(flt!1))]
                  ENDIF(flt!1))
             AND pass(a) /= pas!1}]
        = s1!1
```

Since the IF THEN ELSE is not at the highest level in formula [2], we issue a LIFT-IF command:

```
Rule?: (LIFT-IF 2)
Lifting IF-conditions to the top level,
this simplifies to:
Make_Cancel :

  |-------
[1]    (EXISTS (a: seat_assignment): pass(a) = pas!1 AND member(a, s1!1(flt!1)))
{2}    IF pref_filled(s1!1, flt!1, pref!1) OR pass_on_flight(pas!1, flt!1, s1!1)
         THEN s1!1
           WITH [flt!1 :=
             {a: seat_assignment | member(a, s1!1(flt!1)) AND pass(a) /= pas!1}]
           = s1!1
       ELSE
```

```
            s1!1
              WITH [flt!1 :=
                add((# seat := Next_seat(s1!1, flt!1, pref!1), pass := pas!1
                       #), s1!1(flt!1))]
              WITH [flt!1 :=
                {a: seat_assignment |
                member(a,
                       s1!1
                         WITH [flt!1 :=
                           add((# seat := Next_seat(s1!1, flt!1, pref!1),
                                  pass := pas!1
                                  #), s1!1(flt!1))](flt!1))
                  AND pass(a) /= pas!1}]
              = s1!1
          ENDIF
```

Since the IF THEN ELSE is on the conclusions side of the sequent, it is equivalent to a conjunction. Thus, we use a SPLIT command:

```
Rule?: (SPLIT 2)
Splitting conjunctions,
this yields  2 subgoals:
Make_Cancel.1 :

  |-------
{1}   pref_filled(s1!1, flt!1, pref!1) OR pass_on_flight(pas!1, flt!1, s1!1)
        IMPLIES s1!1
          WITH [flt!1 :=
            {a: seat_assignment | member(a, s1!1(flt!1)) AND pass(a) /= pas!1}]
          = s1!1
[2]   (EXISTS (a: seat_assignment): pass(a) = pas!1 AND member(a, s1!1(flt!1)))
```

We remove the IMPLIES in formula {1} with a FLATTEN command:

```
Rule?: (FLATTEN)
Applying disjunctive simplification to flatten sequent,
this simplifies to:
Make_Cancel.1 :

{-1}   pref_filled(s1!1, flt!1, pref!1) OR pass_on_flight(pas!1, flt!1, s1!1)
  |-------
{1}   s1!1
        WITH [flt!1 :=
          {a: seat_assignment | member(a, s1!1(flt!1)) AND pass(a) /= pas!1}]
        = s1!1
[2]   (EXISTS (a: seat_assignment): pass(a) = pas!1 AND member(a, s1!1(flt!1)))
```

We notice that formula {1} states that two functions are equal[13]. In PVS one proves the equality of functions using extensionality axioms. These are axioms of the form $f = g$ IFF $\forall x : f(x) = g(x)$.

[13] s1!1 is a function[flight -> flight_assignments].

Thus, to prove $f = g$, it is sufficient to prove that $f(x) = g(x)$ for all values of x. In order to bring the appropriate axiom into the sequent, one uses the APPLY-EXTENSIONALITY command:

```
Rule?: (APPLY-EXTENSIONALITY 1)
Rewriting member(a, s1!1(flt!1)) to s1!1(flt!1)(a).
Rewriting member(x,
       s1!1
         WITH [flt!1 :=
           {a: seat_assignment | s1!1(flt!1)(a) AND pass(a) /= pas!1}](x!1))
  to s1!1
    WITH [flt!1 :=
      {a: seat_assignment | s1!1(flt!1)(a) AND pass(a) /= pas!1}](x!1)(x).
Rewriting member(x, s1!1(x!1)) to s1!1(x!1)(x).
Rewriting member(a, s1!1(flt!1)) to s1!1(flt!1)(a).
Rewriting member(a, s1!1(flt!1)) to s1!1(flt!1)(a).
Rewriting member(x,
       s1!1
         WITH [flt!1 :=
           {a: seat_assignment | s1!1(flt!1)(a) AND pass(a) /= pas!1}](x!1))
  to s1!1
    WITH [flt!1 :=
      {a: seat_assignment | s1!1(flt!1)(a) AND pass(a) /= pas!1}](x!1)(x).
Rewriting member(x, s1!1(x!1)) to s1!1(x!1)(x).
Applying extensionality,
this simplifies to:
Make_Cancel.1 :

[-1]   pref_filled(s1!1, flt!1, pref!1) OR pass_on_flight(pas!1, flt!1, s1!1)
  |-------
{1}   s1!1
        WITH [flt!1 :=
          {a: seat_assignment | s1!1(flt!1)(a) AND pass(a) /= pas!1}](x!1)
        = s1!1(x!1)
{2}   s1!1
        WITH [flt!1 :=
          {a: seat_assignment | s1!1(flt!1)(a) AND pass(a) /= pas!1}]
        = s1!1
{3}   (EXISTS (a: seat_assignment): pass(a) = pas!1 AND s1!1(flt!1)(a))
```

Notice that this command has added the formula {1} to the sequent. Usually it is easier to prove formulas like {1} than formulas like [2]. We will not be needing formula [2] any more, so we hide it to keep the clutter down. This is accomplished by the HIDE command:

```
Rule?: (HIDE 2)
Hiding formulas:  2,
this simplifies to:
Make_Cancel.1 :

[-1]   pref_filled(s1!1, flt!1, pref!1) OR pass_on_flight(pas!1, flt!1, s1!1)
```

```
  |-------
[1]   s1!1
        WITH [flt!1 :=
          {a: seat_assignment | s1!1(flt!1)(a) AND pass(a) /= pas!1}](x!1)
        = s1!1(x!1)
[2]   (EXISTS (a: seat_assignment): pass(a) = pas!1 AND s1!1(flt!1)(a))
```

The equality in formula [1] is trivial except for `s1!1(flt!1)`, so we case split on `flt!1 = x!1`:

```
Rule?: (CASE "flt!1 = x!1")
Case splitting on
   flt!1 = x!1,
this yields  2 subgoals:
Make_Cancel.1.1 :

{-1}   flt!1 = x!1
[-2]   pref_filled(s1!1, flt!1, pref!1) OR pass_on_flight(pas!1, flt!1, s1!1)
  |-------
[1]   s1!1
        WITH [flt!1 :=
          {a: seat_assignment | s1!1(flt!1)(a) AND pass(a) /= pas!1}](x!1)
        = s1!1(x!1)
[2]   (EXISTS (a: seat_assignment): pass(a) = pas!1 AND s1!1(flt!1)(a))
```

We simplify with `ASSERT`:

```
Rule?: (ASSERT)
Rewriting member(x, {a: seat_assignment | s1!1(flt!1)(a) AND pass(a) /= pas!1})
  to s1!1(flt!1)(x) AND pass(x) /= pas!1.
Rewriting member(x, s1!1(x!1)) to s1!1(x!1)(x).
Invoking decision procedures,
this simplifies to:
Make_Cancel.1.1 :

[-1]   flt!1 = x!1
[-2]   pref_filled(s1!1, flt!1, pref!1) OR pass_on_flight(pas!1, flt!1, s1!1)
  |-------
{1}   ({a: seat_assignment | s1!1(flt!1)(a) AND pass(a) /= pas!1}) = s1!1(x!1)
[2]   (EXISTS (a: seat_assignment): pass(a) = pas!1 AND s1!1(flt!1)(a))
```

We remember that sets in PVS are just functions into `bool`, so formula {1} involves the equality of two functions. As before we invoke the `APPLY-EXTENSIONALITY` command to introduce the appropriate axiom into the sequent:

```
Rule?: (APPLY-EXTENSIONALITY 1)
Rewriting member(x, {a: seat_assignment | s1!1(flt!1)(a) AND pass(a) /= pas!1})
  to s1!1(flt!1)(x) AND pass(x) /= pas!1.
Rewriting member(x, s1!1(x!1)) to s1!1(x!1)(x).
Rewriting member(x, {a: seat_assignment | s1!1(flt!1)(a) AND pass(a) /= pas!1})
```

```
  to s1!1(flt!1)(x) AND pass(x) /= pas!1.
Rewriting member(x, s1!1(x!1)) to s1!1(x!1)(x).
Applying extensionality,
this simplifies to:
Make_Cancel.1.1 :

[-1]   flt!1 = x!1
[-2]   pref_filled(s1!1, flt!1, pref!1) OR pass_on_flight(pas!1, flt!1, s1!1)
  |-------
{1}   (s1!1(flt!1)(x!2) AND pass(x!2) /= pas!1) = s1!1(x!1)(x!2)
[2]   ({a: seat_assignment | s1!1(flt!1)(a) AND pass(a) /= pas!1}) = s1!1(x!1)
[3]   (EXISTS (a: seat_assignment): pass(a) = pas!1 AND s1!1(flt!1)(a))
```

We won't need formula [2], so we hide it.

```
Rule?: (HIDE 2)
Hiding formulas:  2,
this simplifies to:
Make_Cancel.1.1 :

[-1]   flt!1 = x!1
[-2]   pref_filled(s1!1, flt!1, pref!1) OR pass_on_flight(pas!1, flt!1, s1!1)
  |-------
[1]   (s1!1(flt!1)(x!2) AND pass(x!2) /= pas!1) = s1!1(x!1)(x!2)
[2]   (EXISTS (a: seat_assignment): pass(a) = pas!1 AND s1!1(flt!1)(a))
```

We notice that `s1!1(x!1)` appears in formula [1], while `s1!1[flt!1]` appears in formula [2]. Formula [-1] tells us that `flt!1 = x!1`, so we replace formulas [1] and [2] with [-1]:

```
Rule?: (REPLACE -1 * RL)
Replacing using formula -1,
this simplifies to:
Make_Cancel.1.1 :

[-1]   flt!1 = x!1
[-2]   pref_filled(s1!1, flt!1, pref!1) OR pass_on_flight(pas!1, flt!1, s1!1)
  |-------
{1}   (s1!1(flt!1)(x!2) AND pass(x!2) /= pas!1) = s1!1(flt!1)(x!2)
[2]   (EXISTS (a: seat_assignment): pass(a) = pas!1 AND s1!1(flt!1)(a))
```

To match formula [2] with formula [1], we instantiate formula [2]'s existential quantifier with `x!2`:

```
Rule?: (INST 2 "x!2")
Instantiating the top quantifier in 2 with the terms:
 x!2
this simplifies to:
Make_Cancel.1.1 :

[-1]   flt!1 = x!1
[-2]   pref_filled(s1!1, flt!1, pref!1) OR pass_on_flight(pas!1, flt!1, s1!1)
```

```
  |-------
[1]   (s1!1(flt!1)(x!2) AND pass(x!2) /= pas!1) = s1!1(flt!1)(x!2)
{2}   pass(x!2) = pas!1 AND s1!1(flt!1)(x!2)
```

We now finish off this sequent with GROUND:

```
Rule?: (GROUND)
Applying propositional simplification and decision procedures,

This completes the proof of Make_Cancel.1.1.

Make_Cancel.1.2 :

[-1]   pref_filled(s1!1, flt!1, pref!1) OR pass_on_flight(pas!1, flt!1, s1!1)
  |-------
{1}   flt!1 = x!1
[2]   s1!1
        WITH [flt!1 :=
          {a: seat_assignment | s1!1(flt!1)(a) AND pass(a) /= pas!1}](x!1)
        = s1!1(x!1)
[3]   (EXISTS (a: seat_assignment): pass(a) = pas!1 AND s1!1(flt!1)(a))
```

The prover now turns our attention to Make_Cancel.1.2. We issue an ASSERT:

```
Rule?: (ASSERT)
Invoking decision procedures,

This completes the proof of Make_Cancel.1.2.


This completes the proof of Make_Cancel.1.

Make_Cancel.2 :

  |-------
{1}   NOT
        (pref_filled(s1!1, flt!1, pref!1)
           OR pass_on_flight(pas!1, flt!1, s1!1))
        IMPLIES
        s1!1
          WITH [flt!1 :=
            add((# seat := Next_seat(s1!1, flt!1, pref!1), pass := pas!1
                   #), s1!1(flt!1))]
          WITH [flt!1 :=
            {a: seat_assignment |
            member(a,
                   s1!1
                     WITH [flt!1 :=
                       add((# seat := Next_seat(s1!1, flt!1, pref!1),
```

```
                                        pass := pas!1
                                        #), s1!1(flt!1))](flt!1))
                         AND pass(a) /= pas!1}]
                     = s1!1
  [2]    (EXISTS (a: seat_assignment): pass(a) = pas!1 AND member(a, s1!1(flt!1)))
```

That finishes off Make_Cancel.1.2 and consequently Make_Cancel.1. We are now working on Make_Cancel.2. We issue a GROUND command:

```
Rule?: (GROUND)
Rewriting member(a, s1!1(flt!1)) to s1!1(flt!1)(a).
Rewriting add((# seat := Next_seat(s1!1, flt!1, pref!1), pass := pas!1 #),
   s1!1(flt!1))(a) to (# seat := Next_seat(s1!1, flt!1, pref!1), pass := pas!1 #)
    = a  OR s1!1(flt!1)(a).
Rewriting member(a,
       add((# seat := Next_seat(s1!1, flt!1, pref!1), pass := pas!1 #),
   s1!1(flt!1))) to (# seat := Next_seat(s1!1, flt!1, pref!1), pass := pas!1 #)
    = a  OR s1!1(flt!1)(a).
Rewriting member(a, s1!1(flt!1)) to s1!1(flt!1)(a).
Applying propositional simplification and decision procedures,
this simplifies to:
Make_Cancel.2 :

  |-------
{1}   pref_filled(s1!1, flt!1, pref!1)
{2}   pass_on_flight(pas!1, flt!1, s1!1)
{3}   s1!1
        WITH [flt!1 :=
          add((# seat := Next_seat(s1!1, flt!1, pref!1), pass := pas!1
                 #), s1!1(flt!1))]
        WITH [flt!1 :=
          {a: seat_assignment |
          ((# seat := Next_seat(s1!1, flt!1, pref!1), pass := pas!1 #) = a
             OR s1!1(flt!1)(a))
            AND pass(a) /= pas!1}]
        = s1!1
{4}   (EXISTS (a: seat_assignment): pass(a) = pas!1 AND s1!1(flt!1)(a))
```

Since formula {3} involves the equality of two functions, we issue an APPLY-EXTENSIONALITY command:

```
Rule?: (APPLY-EXTENSIONALITY 3)
Rewriting member(x,
       s1!1
         WITH [flt!1 :=
           add((# seat := Next_seat(s1!1, flt!1, pref!1), pass := pas!1
                  #), s1!1(flt!1))]
         WITH [flt!1 :=
           {a: seat_assignment |
```

```
          ((# seat := Next_seat(s1!1, flt!1, pref!1), pass := pas!1 #) = a
             OR s1!1(flt!1)(a))
            AND pass(a) /= pas!1}](x!1)) to s1!1
  WITH [flt!1 :=
    add((# seat := Next_seat(s1!1, flt!1, pref!1), pass := pas!1
          #), s1!1(flt!1))]
  WITH [flt!1 :=
    {a: seat_assignment |
    ((# seat := Next_seat(s1!1, flt!1, pref!1), pass := pas!1 #) = a
       OR s1!1(flt!1)(a))
      AND pass(a) /= pas!1}](x!1)(x).
Rewriting member(x, s1!1(x!1)) to s1!1(x!1)(x).
Rewriting member(x,
       s1!1
         WITH [flt!1 :=
           add((# seat := Next_seat(s1!1, flt!1, pref!1), pass := pas!1
                 #), s1!1(flt!1))]
         WITH [flt!1 :=
           {a: seat_assignment |
           ((# seat := Next_seat(s1!1, flt!1, pref!1), pass := pas!1 #) = a
              OR s1!1(flt!1)(a))
             AND pass(a) /= pas!1}](x!1)) to s1!1
  WITH [flt!1 :=
    add((# seat := Next_seat(s1!1, flt!1, pref!1), pass := pas!1
          #), s1!1(flt!1))]
  WITH [flt!1 :=
    {a: seat_assignment |
    ((# seat := Next_seat(s1!1, flt!1, pref!1), pass := pas!1 #) = a
       OR s1!1(flt!1)(a))
      AND pass(a) /= pas!1}](x!1)(x).
Rewriting member(x, s1!1(x!1)) to s1!1(x!1)(x).
Applying extensionality,
this simplifies to:
Make_Cancel.2 :

  |-------
{1}   s1!1
        WITH [flt!1 :=
          add((# seat := Next_seat(s1!1, flt!1, pref!1), pass := pas!1
                #), s1!1(flt!1))]
        WITH [flt!1 :=
          {a: seat_assignment |
          ((# seat := Next_seat(s1!1, flt!1, pref!1), pass := pas!1 #) = a
             OR s1!1(flt!1)(a))
            AND pass(a) /= pas!1}](x!1)
        = s1!1(x!1)
[2]   pref_filled(s1!1, flt!1, pref!1)
```

```
[3]   pass_on_flight(pas!1, flt!1, s1!1)
[4]   s1!1
        WITH [flt!1 :=
          add((# seat := Next_seat(s1!1, flt!1, pref!1), pass := pas!1
                #), s1!1(flt!1))]
        WITH [flt!1 :=
          {a: seat_assignment |
          ((# seat := Next_seat(s1!1, flt!1, pref!1), pass := pas!1 #) = a
             OR s1!1(flt!1)(a))
            AND pass(a) /= pas!1}]
        = s1!1
[5]   (EXISTS (a: seat_assignment): pass(a) = pas!1 AND s1!1(flt!1)(a))
```

We hide formula [4]:

```
Rule?: (HIDE 4)
Hiding formulas:  4,
this simplifies to:
Make_Cancel.2 :

  |-------
[1]   s1!1
        WITH [flt!1 :=
          add((# seat := Next_seat(s1!1, flt!1, pref!1), pass := pas!1
                #), s1!1(flt!1))]
        WITH [flt!1 :=
          {a: seat_assignment |
          ((# seat := Next_seat(s1!1, flt!1, pref!1), pass := pas!1 #) = a
             OR s1!1(flt!1)(a))
            AND pass(a) /= pas!1}](x!1)
        = s1!1(x!1)
[2]   pref_filled(s1!1, flt!1, pref!1)
[3]   pass_on_flight(pas!1, flt!1, s1!1)
[4]   (EXISTS (a: seat_assignment): pass(a) = pas!1 AND s1!1(flt!1)(a))
```

Once again the function equality is trivial except for `flt!1 = x!1`, so we case split:

```
Rule?: (CASE "flt!1 = x!1")
Case splitting on
   flt!1 = x!1,
this yields  2 subgoals:
Make_Cancel.2.1 :

{-1}   flt!1 = x!1
  |-------
[1]   s1!1
        WITH [flt!1 :=
          add((# seat := Next_seat(s1!1, flt!1, pref!1), pass := pas!1
                #), s1!1(flt!1))]
```

```
       WITH [flt!1 :=
         {a: seat_assignment |
         ((# seat := Next_seat(s1!1, flt!1, pref!1), pass := pas!1 #) = a
            OR s1!1(flt!1)(a))
           AND pass(a) /= pas!1}](x!1)
       = s1!1(x!1)
[2]   pref_filled(s1!1, flt!1, pref!1)
[3]   pass_on_flight(pas!1, flt!1, s1!1)
[4]   (EXISTS (a: seat_assignment): pass(a) = pas!1 AND s1!1(flt!1)(a))
```

We simplify with ASSERT:

```
Rule?: (ASSERT)
Rewriting member(x,
       {a: seat_assignment |
       ((# seat := Next_seat(s1!1, flt!1, pref!1), pass := pas!1 #) = a
          OR s1!1(flt!1)(a))
         AND pass(a) /= pas!1}) to ((# seat := Next_seat(s1!1, flt!1, pref!1),
            pass := pas!1 #) = x
   OR s1!1(flt!1)(x)) AND pass(x) /= pas!1.
Rewriting member(x, s1!1(x!1)) to s1!1(x!1)(x).
Invoking decision procedures,
this simplifies to:
Make_Cancel.2.1 :

[-1]   flt!1 = x!1
  |-------
{1}   ({a: seat_assignment |
       ((# seat := Next_seat(s1!1, flt!1, pref!1), pass := pas!1 #) = a
          OR s1!1(flt!1)(a))
         AND pass(a) /= pas!1})
        = s1!1(x!1)
[2]   pref_filled(s1!1, flt!1, pref!1)
[3]   pass_on_flight(pas!1, flt!1, s1!1)
[4]   (EXISTS (a: seat_assignment): pass(a) = pas!1 AND s1!1(flt!1)(a))
```

Once again we have a formula that involves the equality of two functions, one of which is a set. Thus, we issue an APPLY-EXTENSIONALITY command followed by the usual HIDE command:

```
Rule?: (APPLY-EXTENSIONALITY 1)
Rewriting member(x,
       {a: seat_assignment |
       ((# seat := Next_seat(s1!1, flt!1, pref!1), pass := pas!1 #) = a
          OR s1!1(flt!1)(a)) AND pass(a) /= pas!1})
  to ((# seat := Next_seat(s1!1, flt!1, pref!1), pass := pas!1 #) = x
   OR s1!1(flt!1)(x)) AND pass(x) /= pas!1.
Rewriting member(x, s1!1(x!1)) to s1!1(x!1)(x).
Rewriting member(x,
       {a: seat_assignment |
```

```
        ((# seat := Next_seat(s1!1, flt!1, pref!1), pass := pas!1 #) = a
           OR s1!1(flt!1)(a)) AND pass(a) /= pas!1})
   to ((# seat := Next_seat(s1!1, flt!1, pref!1), pass := pas!1 #) = x
    OR s1!1(flt!1)(x)) AND pass(x) /= pas!1.
Rewriting member(x, s1!1(x!1)) to s1!1(x!1)(x).
Applying extensionality,
this simplifies to:
Make_Cancel.2.1 :

[-1]   flt!1 = x!1
  |-------
{1}   (((# seat := Next_seat(s1!1, flt!1, pref!1), pass := pas!1 #) = x!2
          OR s1!1(flt!1)(x!2))
        AND pass(x!2) /= pas!1)
       = s1!1(x!1)(x!2)
[2]   ({a: seat_assignment |
       ((# seat := Next_seat(s1!1, flt!1, pref!1), pass := pas!1 #) = a
          OR s1!1(flt!1)(a))
        AND pass(a) /= pas!1})
       = s1!1(x!1)
[3]   pref_filled(s1!1, flt!1, pref!1)
[4]   pass_on_flight(pas!1, flt!1, s1!1)
[5]   (EXISTS (a: seat_assignment): pass(a) = pas!1 AND s1!1(flt!1)(a))

Rule?: (HIDE 2)
Hiding formulas:  2,
this simplifies to:
Make_Cancel.2.1 :

[-1]   flt!1 = x!1
  |-------
[1]   (((# seat := Next_seat(s1!1, flt!1, pref!1), pass := pas!1 #) = x!2
          OR s1!1(flt!1)(x!2))
        AND pass(x!2) /= pas!1)
       = s1!1(x!1)(x!2)
[2]   pref_filled(s1!1, flt!1, pref!1)
[3]   pass_on_flight(pas!1, flt!1, s1!1)
[4]   (EXISTS (a: seat_assignment): pass(a) = pas!1 AND s1!1(flt!1)(a))
```

As in past cases we take advantage of formula [1] using the REPLACE command[14].

```
Rule?: (REPLACE -1 * RL)
Replacing using formula -1,
this simplifies to:
Make_Cancel.2.1 :
```

[14] The PVS decision procedures are powerful enough to prove this sequent and the previous one even if the REPLACE commands are omitted. Nevertheless, often the discovery of a proof is easier when the REPLACE command is used in situations such as these.

```
[-1]   flt!1 = x!1
  |-------
{1}   (((# seat := Next_seat(s1!1, flt!1, pref!1), pass := pas!1 #) = x!2
          OR s1!1(flt!1)(x!2))
         AND pass(x!2) /= pas!1)
        = s1!1(flt!1)(x!2)
[2]   pref_filled(s1!1, flt!1, pref!1)
[3]   pass_on_flight(pas!1, flt!1, s1!1)
[4]   (EXISTS (a: seat_assignment): pass(a) = pas!1 AND s1!1(flt!1)(a))
```

To match the `s1!1(flt!1)(x!2)` in formula [1] with the `s1!1(flt!1)(a)` in formula [4], we instantiate formula [4]:

```
Rule?: (INST 4 "x!2")
Instantiating the top quantifier in 4 with the terms:
 x!2
this simplifies to:
Make_Cancel.2.1 :

[-1]   flt!1 = x!1
  |-------
[1]   (((# seat := Next_seat(s1!1, flt!1, pref!1), pass := pas!1 #) = x!2
          OR s1!1(flt!1)(x!2))
         AND pass(x!2) /= pas!1)
        = s1!1(flt!1)(x!2)
[2]   pref_filled(s1!1, flt!1, pref!1)
[3]   pass_on_flight(pas!1, flt!1, s1!1)
{4}   pass(x!2) = pas!1 AND s1!1(flt!1)(x!2)
```

We have now reached the point where one must know that PVS's decision procedures are not complete for equality over the booleans. Thus, it is necessary to convert the `=` in formula [1] to an `IFF`. This is done using the `IFF` command:

```
Rule?: (IFF 1)
Converting top level boolean equality into IFF form,
Converting equality to IFF,
this simplifies to:
Make_Cancel.2.1 :

[-1]   flt!1 = x!1
  |-------
{1}   ((# seat := Next_seat(s1!1, flt!1, pref!1), pass := pas!1 #) = x!2
         OR s1!1(flt!1)(x!2))
        AND pass(x!2) /= pas!1
        IFF s1!1(flt!1)(x!2)
[2]   pref_filled(s1!1, flt!1, pref!1)
[3]   pass_on_flight(pas!1, flt!1, s1!1)
[4]   pass(x!2) = pas!1 AND s1!1(flt!1)(x!2)
```

Now the decision procedures can finish off this sequent:

```
Rule?: (GROUND)
Applying propositional simplification and decision procedures,

This completes the proof of Make_Cancel.2.1.

Make_Cancel.2.2 :

  |-------
{1}   flt!1 = x!1
[2]   s1!1
        WITH [flt!1 :=
          add((# seat := Next_seat(s1!1, flt!1, pref!1), pass := pas!1
                 #), s1!1(flt!1))]
        WITH [flt!1 :=
          {a: seat_assignment |
          ((# seat := Next_seat(s1!1, flt!1, pref!1), pass := pas!1 #) = a
             OR s1!1(flt!1)(a))
            AND pass(a) /= pas!1}](x!1)
        = s1!1(x!1)
[3]   pref_filled(s1!1, flt!1, pref!1)
[4]   pass_on_flight(pas!1, flt!1, s1!1)
[5]   (EXISTS (a: seat_assignment): pass(a) = pas!1 AND s1!1(flt!1)(a))
```

This completes Make_Cancel.2.1 and we are directed to work on Make_Cancel.2.2. An ASSERT finishes off this branch and the whole proof:

```
Rule? (ASSERT)
Invoking decision procedures,

This completes the proof of Make_Cancel.2.2.


This completes the proof of Make_Cancel.2.

Q.E.D.


Run time  = 49.55 secs.
Real time = 61.22 secs.
```

M-x edit-pr displays the following complete proof:

```
("" (SKOLEM 1
            ("flt!1" "pas!1" "pref!1" "s1!1"))
    (FLATTEN)
    (EXPAND "Cancel_assn")
```

```
(EXPAND "pass_on_flight")
(EXPAND "Make_assn")
(AUTO-REWRITE-THEORY "sets[seat_assignment]")
(LIFT-IF 2)
(SPLIT 2)
(("1" (FLATTEN)
      (APPLY-EXTENSIONALITY 1)
      (HIDE 2)
      (CASE "flt!1 = x!1")
      (("1" (ASSERT)
            (APPLY-EXTENSIONALITY 1)
            (HIDE 2)
            (REPLACE -1 * RL)
            (INST 2 "x!2")
            (GROUND))
        ("2" (ASSERT))))
  ("2" (GROUND)
       (APPLY-EXTENSIONALITY 3)
       (HIDE 4)
       (CASE "flt!1 = x!1")
       (("1" (ASSERT)
             (APPLY-EXTENSIONALITY 1)
             (HIDE 2)
             (REPLACE -1 * RL)
             (INST 4 "x!2")
             (IFF 1)
             (GROUND))
         ("2" (ASSERT)))))
```

We issue a `M-x prt` on the theory. All of the proofs are successful—the system reports:

```
Proof summary for theory ops
    Cancel_assn_inv......................................proved - complete
    MAe..................................................proved - complete
    MAu..................................................proved - complete
    Make_assn_inv........................................proved - complete
    Make_Cancel..........................................proved - complete
    initial_state_inv....................................proved - complete
    Theory totals: 6 formulas, 6 attempted, 6 succeeded.
```

The following putative theorems are left as exercises for the reader:

```
Make_putative: THEOREM NOT pref_filled(s1, flt, pref) =>
        (EXISTS (x: seat_assignment):
          member(x, Make_assn(flt, pas, pref, s1)(flt)) AND pass(x) = pas)

Cancel_putative: THEOREM
        NOT (EXISTS (a: seat_assignment):
           member(a,Cancel_assn(flt,pas,s1)(flt)) AND pass(a) = pas)
```

The ambitious reader should add the following definition to the ops theory:

```
Lookup: function[flight, passenger, assn_state -> [row,position]] =
  (LAMBDA flt, pas, s1:
       seat(epsilon( {a | member(a,s1(flt)) AND pass(a) = pas})))
```

and prove

```
Lookup_putative: THEOREM NOT (pref_filled(s1, flt, pref) OR
                                pass_on_flight(pas,flt,s1)) =>
    meets_pref(aircraft(flt),
               Lookup(flt, pas, Make_assn(flt,pas,pref,s1)),
               pref)
```

4 Summary

A specification of an airline reservation system was formally specified using PVS. A state-machine approach was used to model this system. Two operations were defined and shown to maintain the state invariant. These proofs were accomplished using the PVS prover and discussed in detail. The technique of validating a specification via "putative theorem proving" was also discussed and illustrated in detail.

References

[1] Rushby, John: *Formal Methods and Digital Systems Validation for Airborne Systems.* NASA Contractor Report 4551, 1993.

[2] Shankar, Natarajan; Owre, Sam; and Rushby, John: *PVS Tutorial.* Computer Science Laboratory, SRI International, Menlo Park, CA, Feb. 1993. Also appears in Tutorial Notes, *Formal Methods Europe '93: Industrial-Strength Formal Methods*, pages 357–406, Odense, Denmark, April 1993.

[3] Shankar, N.; Owre, S.; and Rushby, J. M.: *The PVS Proof Checker: A Reference Manual (Beta Release).* Computer Science Laboratory, SRI International, Menlo Park, CA, Feb. 1993.

[4] Owre, S.; Shankar, N.; and Rushby, J. M.: *The PVS Specification Language (Beta Release).* Computer Science Laboratory, SRI International, Menlo Park, CA, Feb. 1993.

[5] Owre, S.; Shankar, N.; and Rushby, J. M.: *User Guide for the PVS Specification and Verification System (Beta Release).* Computer Science Laboratory, SRI International, Menlo Park, CA, Feb. 1993.

[6] Johnson, Sally C.; Holloway, C. Michael; and Butler, Ricky W.: *Second NASA Formal Methods Workshop 1992.* NASA Conference Publication 10110, Nov. 1992.

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. AGENCY USE ONLY *(Leave blank)*

2. REPORT DATE
September 1993

3. REPORT TYPE AND DATES COVERED
Technical Memorandum

4. TITLE AND SUBTITLE
An Elementary Tutorial on Formal Specification and Verification Using PVS

5. FUNDING NUMBERS
WU 505-64-10-13

6. AUTHOR(S)
Ricky W. Butler

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
NASA Langley Research Center
Hampton, VA 23681-0001

8. PERFORMING ORGANIZATION REPORT NUMBER

9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES)
National Aeronautics and Space Administration
Washington, DC 20546-0001

10. SPONSORING / MONITORING AGENCY REPORT NUMBER
NASA TM-108991

11. SUPPLEMENTARY NOTES

12a. DISTRIBUTION / AVAILABILITY STATEMENT
Unclassified - Unlimited

Subject Category 62

12b. DISTRIBUTION CODE

13. ABSTRACT *(Maximum 200 words)*

This paper presents a tutorial on the development of a formal specification and its verification using the Prototype Verification System (PVS). The tutorial presents the formal specification and verification techniques by way of specific example--an airline reservation system. The airline reservation system is modeled as a simple state machine with two basic operations. These operations are shown to preserve a state invariant using the theorem proving capabilities of PVS. The technique of validating a specification via "putative theorem proving" is also discussed and illustrated in detail. This paper is intended for the novice and assumes only some of the basic concepts of logic. A complete description of user inputs and the PVS output is provided and thus it can be effectively used while one is sitting at a computer terminal.

14. SUBJECT TERMS Formal methods; Formal specification; Verification and validation; Theorem provers; Mechanical verification

15. NUMBER OF PAGES
70

16. PRICE CODE
A04

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
Unclassified	Unclassified		

NSN 7540-01-280-5500

Standard Form 298 (Rev. 2-89)
Prescribed by ANSI Std. Z39-18
298-102