NASA Technical Memorandum 108991 (Revised)

An Elementary Tutorial on Formal Specification and Verification Using PVS 2

Ricky W. Butler

September 1993 (revised June 1995)

NASA
National Aeronautics and
Space Administration

Langley Research Center
Hampton, VA 23681

Abstract

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 a 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. PVS is free and can be retrieved via anonymous FTP. For information about how to obtain and install PVS, see World Wide Web at http://www.csl.sri.com/sri-csl-pvs.html.

KEY WORDS: formal methods, formal specification, verification & validation, theorem provers, PVS, mechanical verification

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. This tutorial was revised in June 1995 to conform with the latest version of PVS, PVS version 2.0. PVS is free and can be retrieved via anonymous FTP. For information about how to obtain and install PVS, see World Wide Web at http://www.csl.sri.com/sri-csl-pvs.html.

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 `revised-specs.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

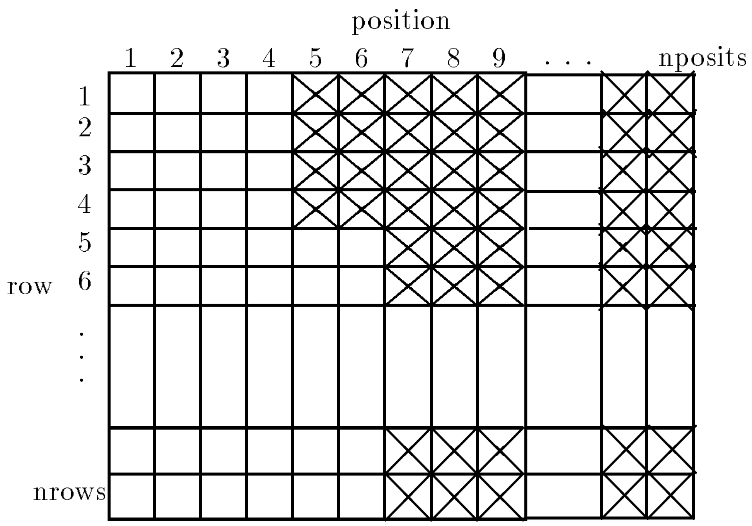

Figure 1: Model of Seating Arrangement In An Airplane

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 3 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:

[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 specification.

```
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.

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:

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

This defines a type that represents the domain of all possible databases. A particular database can be represented by declaring a variable of this type:

```
db: VAR flt_db
```

Initially, each flight has no assignments:

```
initial_state(flt: flight): flight_assignments = 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]

  flt_db: TYPE = function[flight -> flight_assignments]

  initial_state(flt: flight): flight_assignments = 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,db)`, which cancels the seat assignment for a passenger, `pas`, on flight `flt` in database `db`:

```
  a: VAR seat_assignment
  Cancel_assn(flt: flight, pas: passenger, db: flt_db): flt_db =
     db WITH [(flt) := {a | member(a,db(flt)) AND pass(a) /= pas}]
```

The declaration of the function begins with its formal parameter list. Each argument is listed with its type. The return type of the function is given after the parameter list. 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,db)`, which makes a seat assignment, if possible, for passenger `pas` on flight `flt` in database `db`. 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

[2]The resulting function is not different if f(1) = y. Formally `f WITH [(1) := y](x) = IF x = 1 THEN y ELSE f(x) ENDIF`

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

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

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:

```
seat: VAR [row,position]
pref_filled(db: flt_db, flt: flight, pref: preference): bool =
   (FORALL seat: meets_pref(aircraft(flt), seat, pref)
              IMPLIES (EXISTS a: member(a, db(flt)) AND seat(a) = seat))
```

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

```
pass_on_flight(pas: passenger, flt: flight, db: flt_db): bool =
            (EXISTS a: pass(a) = pas AND member(a,db(flt)))
```

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

```
Make_assn(flt:flight, pas:passenger, pref:preference, db:flt_db): flt_db =
      IF pref_filled(db, flt, pref) OR
         pass_on_flight(pas,flt,db) THEN db
      ELSE
         (LET a = (# seat := Next_seat(db,flt,pref), pass := pas #) IN
            db WITH [(flt) := add(a, db(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, db(flt))`, which creates a new set by adding the element `a` to the set `db(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[flt_db, 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:

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

```
db:   VAR flt_db
flt:  VAR flight
pref: VAR preference
Next_seat_ax: AXIOM NOT pref_filled(db, flt, pref) IMPLIES
      seat_exists(aircraft(flt),Next_seat(db,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`. This axiom makes use of several variables. These will be used in subsequent PVS declarations.

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,

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

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,db)(flt)) AND pass(a) = pas)

Make_putative: THEOREM NOT pref_filled(db, flt, pref) =>
        (EXISTS (x: seat_assignment):
          member(x, Make_assn(flt, pas, pref, db)(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. Assigning more than one passenger to a single seat

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

```
existence(db: flt_db): bool =
     (FORALL a,flt: member(a, db(flt)) IMPLIES
     seat_exists(aircraft(flt), seat(a)))
```

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

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

```
uniqueness(db: flt_db): bool =
    (FORALL a,b,flt: member(a, db(flt)) AND member(b, db(flt))
             AND pass(a) = pass(b) IMPLIES a = b)
```

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

```
one_per_seat(db: flt_db): bool =
    (FORALL a,b,flt: member(a, db(flt)) AND member(b, db(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:

```
db_invariant(db: flt_db): bool = existence(db) AND uniqueness(db)
```

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

  a:  VAR seat_assignment
  Cancel_assn(flt: flight, pas: passenger, db: flt_db): flt_db =
     db WITH [(flt) := {a | member(a,db(flt)) AND pass(a) /= pas}]

  seat: VAR [row,position]
  pref_filled(db: flt_db, flt: flight, pref: preference): bool =
     (FORALL seat: meets_pref(aircraft(flt), seat, pref)
   IMPLIES (EXISTS a: member(a, db(flt)) AND seat(a) = seat))

  pass_on_flight(pas: passenger, flt: flight, db: flt_db): bool =
              (EXISTS a: pass(a) = pas AND member(a,db(flt)))

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

  Make_assn(flt:flight, pas:passenger, pref:preference, db:flt_db): flt_db =
        IF pref_filled(db, flt, pref) OR
           pass_on_flight(pas,flt,db) THEN db
        ELSE
   (LET a = (# seat := Next_seat(db,flt,pref),
       pass := pas #) IN
      db WITH [(flt) := add(a, db(flt))])
        ENDIF
```

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

```
% ========================================================================
%                            Variable Declarations
% ========================================================================


  flt:  VAR flight
  pas:  VAR passenger
  db:   VAR flt_db
  b:  VAR seat_assignment
  pref: VAR preference

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


% ========================================================================
%                   Putative Theorems
% ========================================================================

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

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

  existence(db: flt_db): bool =
       (FORALL a,flt: member(a, db(flt)) IMPLIES
       seat_exists(aircraft(flt), seat(a)))

  uniqueness(db: flt_db): bool =
      (FORALL a,b,flt: member(a, db(flt)) AND member(b, db(flt))
               AND pass(a) = pass(b) IMPLIES a = b)

  one_per_seat(db: flt_db): bool =
      (FORALL a,b,flt: member(a, db(flt)) AND member(b, db(flt))
               AND seat(a) = seat(b) IMPLIES a = b)


  db_invariant(db: flt_db): bool = existence(db) AND uniqueness(db)


% ========================================================================
%                            THEOREMS
% ========================================================================
```

```
 Cancel_db_inv: THEOREM db_invariant(db)
IMPLIES db_invariant(Cancel_assn(flt,pas,db))

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

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

 Make_db_inv: THEOREM db_invariant(db) =>
  db_invariant(Make_assn(flt,pas,pref,db))

 initial_state_inv: THEOREM db_invariant(initial_state)
```

When we issue the `M-x tc` command we notice that the system responds `ops typechecked: 1 TCCs, 0 Proved, 0 subsumed, 1 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 (line 24) for
    % Next_seat: FUNCTION[flt_db, flight, preference -> [row, position]]
    %
  % unfinished
Next_seat_TCC1: OBLIGATION
      (EXISTS (x: [[flt_db, flight, preference] -> [row, position]]): TRUE);
```

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 (x: [[flt_db, flight, preference] -> [row, position]]): TRUE)

Rule?
```

The system wants us to prove that there exists a function from the triple `[flt_db, flight, preference]` to the ordered pair `[row, position]`. The constant function that returns the ordered pair `(1,1)` should suffice, but we have not yet defined such a function. Nevertheless, we can introduce an unnamed function using the LAMBDA notation available in PVS:

```
      (LAMBDA (x: [flt_db, flight, preference]): (1,1))
```

The `LAMBDA` keyword merely indicates that a function without a name is being defined. The expressions immediately after a `LAMBDA` define the parameters of the function. In this case, there is one parameter `x` of type `[flt_db, flight, preference]`. The body of the function follows the ":". In this case the body is `(1,1)` which is the contant ordered pair that indicates that `row` = 1 and position = 1. Now to tell the prover to use this function in formula {1}, we use the INST command as follows:

```
Rule? (INST 1 "(LAMBDA (x: [flt_db, flight, preference]): (1,1))")
Instantiating the top quantifier in 1 with the terms:
 (LAMBDA (x: [flt_db, flight, preference]): (1,1)),
this yields  2 subgoals:
Next_seat_TCC1.1 (TCC):

  |-------
{1}   1 <= nposits
```

The prover tells us that we now have two subgoals to prove. The first one named Next_seat_TCC1.1 is displayed for us, Since this is a trivial result that follows directly from the declaration of nposits, we issue an ASSERT command:

```
Rule? (assert)
Simplifying, rewriting, and recording with decision procedures,

This completes the proof of Next_seat_TCC1.1.

Next_seat_TCC1.2 (TCC):

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

The prover tells us that Next_seat_TCC1.1 is complete and displays the other subgoal. We issue ASSERT again, and we receive:

```
Rule? (assert)
Simplifying, rewriting, and recording with decision procedures,

This completes the proof of Next_seat_TCC1.2.

Q.E.D.


Run time  = 1.77 secs.
Real time = 25.04 secs.
```

The prover produces Q.E.D. which informs us that we are done.

3 Formal Verifications

In this section we will walk-through the mechanical verification of the invariant properties discussed previously. This will be done using the basic commands of PVS and not the more powerful strategies. Although skilled PVS users would not approach the proofs in the manner presented here, working through these proofs can be an excellent learning experience for the novice. Once the basic commands have been mastered, the reader should continue with the following more advanced tutorial that is based upon the same example used in this paper:

> J. M. Rushby and D. W. J. Stringer-Calvert "A Less Elementary Tutorial for the PVS Specification and Verification System", SRI International Technical Report CSL-95-10, June 1995.

The more advanced tutorial presents proofs of all of the theorems and lemmas using no more than two or three PVS commands.

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 db_invariant(db) =>
                           db_invariant(Cancel_assn(flt,pas,db))

Make_assn_inv: THEOREM db_invariant(db) =>
                         db_invariant(Make_assn(flt,pas,pref,db))
```

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 [6].

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 (db: flt_db, flt: flight, pas: passenger):
         db_invariant(db) IMPLIES db_invariant(Cancel_assn(flt, pas, db)))
```

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

[6] All of these theorems can be proved using only a few of the more powerful PVS strategies. See

J. M. Rushby and D. W. J. Stringer-Calvert "A Less Elementary Tutorial for the PVS Specification and Verification System", SRI International Technical Report CSL-95-10, June 1995.

formula as a tautology 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. However, the PVS language is case-sensitive and thus identifiers within quotes must be the same as they appear in the specification. The abbreviations for the commands (e.g. TAB E) are case-sensitive.

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 instantiation. Some situations require skolemization and others require instantiation. In this case we need to skolemize formula [1][7]. In PVS this is accomplished using the `SKOSIMP*` command (TAB *):

```
Rule? (SKOSIMP*)
Repeatedly Skolemizing and flattening,
this simplifies to:
Cancel_assn_inv :

{-1}   db_invariant(db!1)
  |-------
{1}   db_invariant(Cancel_assn(flt!1, pas!1, db!1))
```

To save the user from excessive amounts of typing, the PVS system provides abbreviated commands. Thus, the user could have typed TAB * which is the abbreviation for `SKOSIMP*`. We will follow the command name with the abbreviation in parentheses as an aid throughout this paper. Notice that the system has replaced the variable `db` with a new constant `db!1`. Similarly, the variables `pas` and `flt` have been replaced by `pas!1` and `flt!1`, respectively[8].

Clearly, no progress can be made until the meaning of `db_invariant` is exposed to the prover. This is done through use of the `EXPAND` command (TAB e), i.e. `(EXPAND "db_invariant")`[9]. The system responds as follows:

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

{-1}   existence(db!1) AND uniqueness(db!1)
  |-------
{1}   existence(Cancel_assn(flt!1, pas!1, db!1))
          AND uniqueness(Cancel_assn(flt!1, pas!1, db!1))
```

[7]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.

[8]This may not have been your first choice for names, but it has the advantage that the name of the original quantified variable is easily retrieved from the skolem name.

[9]The TAB e command is special in that the user moves the cursor to point to an instance of the identifier that is to be expanded in the current sequent. This is done prior to typing TAB e. This saves the user from having to type in the identifier.

The conjunction in formula `{-1}` can be broken into two separate formulas with the `FLATTEN` command (or TAB f):

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

{-1}   existence(db!1)
{-2}   uniqueness(db!1)
  |-------
[1]   existence(Cancel_assn(flt!1, pas!1, db!1))
          AND uniqueness(Cancel_assn(flt!1, pas!1, db!1))
```

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. The curly braces indicate recently changed formulas whereas the square brackets indicate unchanged formulas. In this instance we have:

$$\{-1\} \wedge \{-2\} \longrightarrow [1]$$

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 (TAB s):

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

{-1}   existence(db!1)
{-2}   uniqueness(db!1)
  |-------
{1}   existence(Cancel_assn(flt!1, pas!1, db!1))
```

The "1" in the command indicates that it should only be applied to formula 1. The system responds with `Splitting conjunctions, this yields 2 subgoals:` and presents us with the first subgoal.

We then proceed to expand with definitions of `existence` and `Cancel_assn`:

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

```
{-1}   (FORALL (a: [# pass: passenger, seat: [row, position] #]), (flt: flight):
         member(a, db!1(flt)) IMPLIES seat_exists(aircraft(flt), seat(a)))
[-2]   uniqueness(db!1)
  |-------
{1}   (FORALL (a: [# pass: passenger, seat: [row, position] #]), (flt: flight):
         member(a, Cancel_assn(flt!1, pas!1, db!1)(flt))
             IMPLIES seat_exists(aircraft(flt), seat(a)))

Rule? (EXPAND "Cancel_assn")
Expanding the definition of Cancel_assn,
this simplifies to:
Cancel_assn_inv.1 :

[-1]   (FORALL (a: [# pass: passenger, seat: [row, position] #]), (flt: flight):
         member(a, db!1(flt)) IMPLIES seat_exists(aircraft(flt), seat(a)))
[-2]   uniqueness(db!1)
  |-------
{1}   (FORALL (a: [# pass: passenger, seat: [row, position] #]), (flt: flight):
         member(a,
                db!1
                  WITH [(flt!1) :=
                          {a:
                            [# pass: passenger,
                               seat: [row, position] #]
                          |
                          member(a, db!1(flt!1))
                              AND pass(a) /= pas!1}](flt))
             IMPLIES seat_exists(aircraft(flt), seat(a)))
```

We note that the function `member` appears in several places in the formula. Although this function is defined in the PVS prelude[10], 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 :

{-1}   (FORALL (a: [# pass: passenger, seat: [row, position] #]), (flt: flight):
         db!1(flt)(a) IMPLIES seat_exists(aircraft(flt), seat(a)))
[-2]   uniqueness(db!1)
  |-------
{1}   (FORALL (a: [# pass: passenger, seat: [row, position] #]), (flt: flight):
         db!1
           WITH [(flt!1) :=
                   {a:
                     [# pass: passenger,
```

[10]This can be inspected by typing `M-x vpf` or `M-x vpt "sets"`

```
                        seat: [row, position] #]
                   |
                   db!1(flt!1)(a)
                       AND pass(a) /= pas!1}](flt)(a)
             IMPLIES seat_exists(aircraft(flt), seat(a)))
```

Notice that member(a,Db(flt)) has been changed to Db(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[11]. This boolean-valued function is true only for members of the set, i.e. S(x) is true if and only if x ∈ S.

Since the formula {1} once again has a FORALL quantifier below the line, we issue a SKOSIMP* command (TAB *) to eliminate it:

```
Rule? (SKOSIMP*)
Repeatedly Skolemizing and flattening,
this simplifies to:
Cancel_assn_inv.1 :

{-1}   db!1
        WITH [(flt!1) :=
                {a:
                  [# pass: passenger,
                     seat: [row, position] #]
                |
                db!1(flt!1)(a)
                    AND pass(a) /= pas!1}](flt!2)(a!1)
[-2]   (FORALL (a: [# pass: passenger, seat: [row, position] #]), (flt: flight):
         db!1(flt)(a) IMPLIES seat_exists(aircraft(flt), seat(a)))
[-3]   uniqueness(db!1)
  |-------
{1}   seat_exists(aircraft(flt!2), seat(a!1))
```

The WITH statement in formula {-1} can be reduced to an equivalent IF-THEN-ELSE using a LIFT-IF command (TAB l):

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

{-1}   IF flt!1 = flt!2
        THEN
        ({a: [# pass: passenger, seat: [row, position] #]
         | db!1(flt!1)(a) AND pass(a) /= pas!1})(a!1)
      ELSE db!1(flt!2)(a!1)
```

[11] This approach is feasible in a higher-order logic and can be shown to be sound.

```
      ENDIF
[-2]   (FORALL (a: [# pass: passenger, seat: [row, position] #]), (flt: flight):
         db!1(flt)(a) IMPLIES seat_exists(aircraft(flt), seat(a)))
[-3]   uniqueness(db!1)
  |-------
[1]   seat_exists(aircraft(flt!2), seat(a!1))
```

Formula [1] requires us to show that seat(a!1) exists on the aircraft associated with flt!2. Formula [-2] contains the function seat_exists, but it has a FORALL quantifier. If we substitute a!1 and flt!2 for the universal (i.e. FORALL) variables) it will match [1]. We can make this substitution (or instantiation) using the INST command (TAB i):

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

[-1]   IF flt!1 = flt!2
        THEN
        ({a: [# pass: passenger, seat: [row, position] #]
         | db!1(flt!1)(a) AND pass(a) /= pas!1})(a!1)
      ELSE db!1(flt!2)(a!1)
      ENDIF
{-2}   db!1(flt!2)(a!1) IMPLIES seat_exists(aircraft(flt!2), seat(a!1))
[-3]   uniqueness(db!1)
  |-------
[1]   seat_exists(aircraft(flt!2), seat(a!1))
```

Notice that the values to be substituted are enclosed in quotes.

We now issue the GROUND command (TAB g). 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, GROUND will automatically finish off the proof.

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

This completes the proof of Cancel_assn_inv.1.

Cancel_assn_inv.2 :

[-1]   existence(db!1)
[-2]   uniqueness(db!1)
  |-------
{1}   uniqueness(Cancel_assn(flt!1, pas!1, db!1))
```

The GROUND command finishes of the first subgoal and the system displays the other subgoal. As with the other subgoal we need to expand the definitions of Cancel_assn and uniqueness:

```
Rule? (EXPAND "Cancel_assn" )
Expanding the definition of Cancel_assn,
this simplifies to:
Cancel_assn_inv.2 :

[-1]   existence(db!1)
[-2]   uniqueness(db!1)
  |-------
{1}   uniqueness(db!1
                 WITH [(flt!1) :=
                         {a:
                           [# pass: passenger,
                              seat: [row, position] #]
                         |
                         member(a, db!1(flt!1))
                             AND pass(a) /= pas!1}])

Rule? (EXPAND "uniqueness" )
Expanding the definition of uniqueness,
this simplifies to:
Cancel_assn_inv.2 :

[-1]   existence(db!1)
{-2}   (FORALL (a: [# pass: passenger, seat: [row, position] #]),
       (b: [# pass: passenger, seat: [row, position] #]), (flt: flight):
         member(a, db!1(flt)) AND member(b, db!1(flt)) AND pass(a) = pass(b)
             IMPLIES a = b)
  |-------
{1}   (FORALL (a: [# pass: passenger, seat: [row, position] #]),
       (b: [# pass: passenger, seat: [row, position] #]), (flt: flight):
         member(a,
                db!1
                  WITH [(flt!1) :=
                         {a:
                           [# pass: passenger,
                              seat: [row, position] #]
                         |
                         member(a, db!1(flt!1))
                             AND pass(a) /= pas!1}](flt))
             AND
           member(b,
                  db!1
                    WITH [(flt!1) :=
                           {a:
                             [# pass: passenger,
```

```
                                  seat: [row, position] #]
                              |
                              member(a, db!1(flt!1))
                                  AND pass(a)
                                    /= pas!1}](flt))
               AND pass(a) = pass(b)
             IMPLIES a = b)
```

We now issue another SKOSIMP* (TAB *) to remove the FORALL in formula [1]:

```
Rule? (SKOSIMP*)
Repeatedly Skolemizing and flattening,
this simplifies to:
Cancel_assn_inv.2 :

{-1}   member(a!1,
             db!1
               WITH [(flt!1) :=
                       {a:
                         [# pass: passenger,
                            seat: [row, position] #]
                       |
                       member(a, db!1(flt!1))
                           AND pass(a) /= pas!1}](flt!2))
{-2}   member(b!1,
             db!1
               WITH [(flt!1) :=
                       {a:
                         [# pass: passenger,
                            seat: [row, position] #]
                       |
                       member(a, db!1(flt!1))
                           AND pass(a) /= pas!1}](flt!2))
{-3}   pass(a!1) = pass(b!1)
[-4]   existence(db!1)
[-5]   (FORALL (a: [# pass: passenger, seat: [row, position] #]),
       (b: [# pass: passenger, seat: [row, position] #]), (flt: flight):
         member(a, db!1(flt)) AND member(b, db!1(flt)) AND pass(a) = pass(b)
             IMPLIES a = b)
  |-------
{1}   a!1 = b!1
```

We need to expand member:

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

```
Cancel_assn_inv.2 :

{-1}   db!1
        WITH [(flt!1) :=
                {a:
                  [# pass: passenger,
                     seat: [row, position] #]
                |
                db!1(flt!1)(a)
                    AND pass(a) /= pas!1}](flt!2)(a!1)
{-2}   db!1
        WITH [(flt!1) :=
                {a:
                  [# pass: passenger,
                     seat: [row, position] #]
                |
                db!1(flt!1)(a)
                    AND pass(a) /= pas!1}](flt!2)(b!1)
[-3]   pass(a!1) = pass(b!1)
[-4]   existence(db!1)
{-5}   (FORALL (a: [# pass: passenger, seat: [row, position] #]),
       (b: [# pass: passenger, seat: [row, position] #]), (flt: flight):
         db!1(flt)(a) AND db!1(flt)(b) AND pass(a) = pass(b) IMPLIES a = b)
  |-------
[1]   a!1 = b!1
```

To replace the WITH statements with the corresponding IF-THEN-ELSE statement, we issue a LIFT-IF (TAB l):

```
{-1}   IF flt!1 = flt!2
        THEN
        ({a: [# pass: passenger, seat: [row, position] #]
         | db!1(flt!1)(a) AND pass(a) /= pas!1})(a!1)
      ELSE db!1(flt!2)(a!1)
      ENDIF
{-2}   IF flt!1 = flt!2
        THEN
        ({a: [# pass: passenger, seat: [row, position] #]
         | db!1(flt!1)(a) AND pass(a) /= pas!1})(b!1)
      ELSE db!1(flt!2)(b!1)
      ENDIF
[-3]   pass(a!1) = pass(b!1)
[-4]   existence(db!1)
[-5]   (FORALL (a: [# pass: passenger, seat: [row, position] #]),
       (b: [# pass: passenger, seat: [row, position] #]), (flt: flight):
         db!1(flt)(a) AND db!1(flt)(b) AND pass(a) = pass(b) IMPLIES a = b)
```

```
  |-------
[1]   a!1 = b!1
```

We see that formula [-5] contains the term a=b. Thus, we want to instantiate the variables a and b with a!1 and b!1. PVS provides a heuristic matching capability that often finds the substitution we want. We decide to try it here. Therefore we issue the INST? (TAB ?) command:

```
Rule? (INST? )

Found substitution:
b gets b!1,
a gets a!1,
Instantiating quantified variables,
this simplifies to:
Cancel_assn_inv.2 :

[-1]   IF flt!1 = flt!2
        THEN
        ({a: [# pass: passenger, seat: [row, position] #]
         | db!1(flt!1)(a) AND pass(a) /= pas!1})(a!1)
      ELSE db!1(flt!2)(a!1)
      ENDIF
[-2]   IF flt!1 = flt!2
        THEN
        ({a: [# pass: passenger, seat: [row, position] #]
         | db!1(flt!1)(a) AND pass(a) /= pas!1})(b!1)
      ELSE db!1(flt!2)(b!1)
      ENDIF
[-3]   pass(a!1) = pass(b!1)
[-4]   existence(db!1)
{-5}   (FORALL (flt: flight):
         db!1(flt)(a!1) AND db!1(flt)(b!1) AND pass(a!1) = pass(b!1)
             IMPLIES a!1 = b!1)
  |-------
[1]   a!1 = b!1
```

and we see that it gets exactly the substitutions that we desired. Nevertheless, formula [-5] still contains a universal quantifier, so we issue another INST? (TAB ?) command:

```
Rule? (INST? )
Found substitution:
flt gets flt!2,
Instantiating quantified variables,
this simplifies to:
Cancel_assn_inv.2 :

[-1]   IF flt!1 = flt!2
```

```
          THEN
          ({a: [# pass: passenger, seat: [row, position] #]
           | db!1(flt!1)(a) AND pass(a) /= pas!1})(a!1)
        ELSE db!1(flt!2)(a!1)
        ENDIF
[-2]    IF flt!1 = flt!2
          THEN
          ({a: [# pass: passenger, seat: [row, position] #]
           | db!1(flt!1)(a) AND pass(a) /= pas!1})(b!1)
        ELSE db!1(flt!2)(b!1)
        ENDIF
[-3]    pass(a!1) = pass(b!1)
[-4]    existence(db!1)
{-5}    db!1(flt!2)(a!1) AND db!1(flt!2)(b!1) AND pass(a!1) = pass(b!1)
           IMPLIES a!1 = b!1
  |-------
[1]   a!1 = b!1
```

Now we issue the GROUND command (TAB g):

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

This completes the proof of Cancel_assn_inv.2.

Q.E.D.


Run time  = 8.64 secs.
Real time = 718.50 secs.

Wrote proof file /airlab/home/rwb/fm/pvs/tutorial-reservation-sys/pvs2/ops.prf
```

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:

```
(""
 (SKOSIMP*)
 (EXPAND "db_invariant")
 (FLATTEN)
 (SPLIT 1)
 (("1"
   (EXPAND "existence")
   (EXPAND "Cancel_assn")
   (EXPAND "member")
   (SKOSIMP*)
   (LIFT-IF -1)
   (INST -2 "a!1" "flt!2")
```

```
 (GROUND))
("2"
 (EXPAND "Cancel_assn")
 (EXPAND "uniqueness")
 (SKOSIMP*)
 (EXPAND "member")
 (LIFT-IF)
 (INST?)
 (INST?)
 (GROUND))))
```

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(db) =>
                           assn_invariant(Make_assn(flt,pas,pref,db))
```

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 [12]:

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

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

Then, we will prove Make_assn_inv from these. The order of the proofs is not critical[13].

3.2.1 Proof of MAe

We begin with MAe

```
MAe :

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

As in the previous proof, we need to eliminate the universal quantifier by skolemization:

[12] This is not necessary for this proof, because the same result can be accomplished with SPLIT; however, this will enable us to illustrate the REWRITE command.

[13] However, many times it is valuable first to prove that the main theorem follows from the lemmas so that one does not prove a useless lemma.

```
Rule? (SKOSIMP*)
Repeatedly Skolemizing and flattening,
this simplifies to:
MAe :

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

Next, we expand the definition of `existence`:

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

{-1}   (FORALL (a: [# pass: passenger, seat: [row, position] #]), (flt: flight):
         member(a, db!1(flt)) IMPLIES seat_exists(aircraft(flt), seat(a)))
  |-------
{1}   (FORALL (a: [# pass: passenger, seat: [row, position] #]), (flt: flight):
         member(a, Make_assn(flt!1, pas!1, pref!1, db!1)(flt))
             IMPLIES seat_exists(aircraft(flt), seat(a)))
```

We expand the definition of `Make_assn`:

```
(expand "Make_assn" )
Expanding the definition of Make_assn,
this simplifies to:
MAe :

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

```
 (AUTO-REWRITE "member")

Installing rewrite rule member
Installing rewrite rule member
Installing automatic rewrites:
  member
this simplifies to:
MAe :

[-1]   (FORALL (a: [# pass: passenger, seat: [row, position] #]), (flt: flight):
         member(a, db!1(flt)) IMPLIES seat_exists(aircraft(flt), seat(a)))
  |-------
[1]   (FORALL (a: [# pass: passenger, seat: [row, position] #]), (flt: flight):
         member(a,
                IF pref_filled(db!1, flt!1, pref!1)
                        OR pass_on_flight(pas!1, flt!1, db!1) THEN db!1(flt)
                ELSE db!1
                    WITH [(flt!1) :=
                            add((# seat :=
                                     Next_seat(db!1,
                                               flt!1, pref!1),
                                   pass := pas!1 #),
                                db!1(flt!1))](flt)
                ENDIF)
             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 (TAB *) to eliminate the universal quantifiers in formula [1]:

```
Rule? (SKOSIMP*)

Repeatedly Skolemizing and flattening,
this simplifies to:
MAe :

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

```
                                          flt!1, pref!1),
                                  pass := pas!1 #),
                                db!1(flt!1))](flt!2)
             ENDIF)
  |-------
{1}   seat_exists(aircraft(flt!2), seat(a!1))
```

The universal quantifier in `{-1}` must be removed by instantiation. 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`. We try `INST?` (TAB ?):

```
Rule? (INST? )
Found substitution:
a gets a!1,
flt gets flt!2,
Instantiating quantified variables,
this simplifies to:
MAe :

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

It finds the right substitutions, so we simplify with `ASSERT`:

```
Rule? (ASSERT)

member rewrites member(a!1, db!1(flt!2))
  to db!1(flt!2)(a!1)
member rewrites
  member(a!1,
         IF pref_filled(db!1, flt!1, pref!1)
                OR pass_on_flight(pas!1, flt!1, db!1) THEN db!1(flt!2)
         ELSE db!1
             WITH [(flt!1) :=
                     add((# seat :=
```

```
                              Next_seat(db!1,
                                        flt!1, pref!1),
                            pass := pas!1 #),
                         db!1(flt!1))](flt!2)
       ENDIF)
  to IF pref_filled(db!1, flt!1, pref!1)
             OR pass_on_flight(pas!1, flt!1, db!1) THEN db!1(flt!2)(a!1)
      ELSE
        db!1
          WITH [(flt!1) :=
                  add((# seat :=
                           Next_seat(db!1,
                                     flt!1, pref!1),
                         pass := pas!1 #),
                      db!1(flt!1))](flt!2)(a!1)
      ENDIF
Simplifying, rewriting, and recording with decision procedures,
this simplifies to:
MAe :

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

We notice that the rewrites of `member` take place at this time. We now notice that we have an `IF THEN ELSE` structure in formula {-1}. We decide to split into two subgoals based on the IF expression using the `SPLIT` command (TAB s):

```
Rule? (SPLIT -1)

Splitting conjunctions,
this yields  2 subgoals:
MAe.1 :

{-1}   (pref_filled(db!1, flt!1, pref!1) OR pass_on_flight(pas!1, flt!1, db!1))
          AND db!1(flt!2)(a!1)
```

```
  |-------
[1]   db!1(flt!2)(a!1)
[2]   seat_exists(aircraft(flt!2), seat(a!1))
```

We issue the GROUND command (TAB g) to finish off the proof of the first subgoal:

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

This completes the proof of MAe.1.

MAe.2 :

{-1}   NOT
        (pref_filled(db!1, flt!1, pref!1)
             OR pass_on_flight(pas!1, flt!1, db!1))
          AND
        db!1
          WITH [(flt!1) :=
                  add((# seat :=
                           Next_seat(db!1,
                                     flt!1, pref!1),
                         pass := pas!1 #),
                      db!1(flt!1))](flt!2)(a!1)
  |-------
[1]   db!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" )
Expanding the definition of add,
this simplifies to:
MAe.2 :

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

```
                    (# seat := Next_seat(db!1, flt!1, pref!1),
                       pass := pas!1 #)
                        = y
                        OR member(y, db!1(flt!1))}](flt!2)(a!1)
  |-------
[1]   db!1(flt!2)(a!1)
[2]   seat_exists(aircraft(flt!2), seat(a!1))
```

The presence of a conjunction (i.e. AND) on the negative side of the sequent (e.g. {-1}) suggests the need for a FLATTEN to simplify the sequent:

```
Rule? (FLATTEN)

Applying disjunctive simplification to flatten sequent,
this simplifies to:
MAe.2 :

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

To simplify the WITH expression of {-1} we issue a LIFT-IF (TAB l) followed by an GROUND (TAB g):

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

{-1}   IF flt!1 = flt!2
        THEN
        ({y: [# pass: passenger, seat: [row, position] #] |
          (# seat := Next_seat(db!1, flt!1, pref!1), pass := pas!1 #) = y
              OR member(y, db!1(flt!1))})(a!1)
      ELSE db!1(flt!2)(a!1)
```

```
      ENDIF
  |-------
[1]   pref_filled(db!1, flt!1, pref!1)
[2]   pass_on_flight(pas!1, flt!1, db!1)
[3]   db!1(flt!2)(a!1)
[4]   seat_exists(aircraft(flt!2), seat(a!1))

Rule? (GROUND)
member rewrites member(y, db!1(flt!1))
  to db!1(flt!1)(y)
Applying propositional simplification and decision procedures,
this simplifies to:
MAe.2 :

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

There is nothing in the antecedent formula that will make `[1]`, `[2]`, `[3]` or `[4]` true. Formula `[4]` asserts that the seat determined by `seat(a!1)` actually exists. But formula `{-2}` 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 (TAB L):

```
Rule? (LEMMA "Next_seat_ax")

Applying Next_seat_ax where
this simplifies to:
MAe.2 :

{-1}   (FORALL (db: flt_db, flt: flight, pref: preference):
         NOT pref_filled(db, flt, pref)
             IMPLIES seat_exists(aircraft(flt), Next_seat(db, flt, pref)))
[-2]   flt!1 = flt!2
[-3]   (# seat := Next_seat(db!1, flt!1, pref!1), pass := pas!1 #) = a!1
  |-------
[1]   pref_filled(db!1, flt!1, pref!1)
[2]   pass_on_flight(pas!1, flt!1, db!1)
[3]   db!1(flt!2)(a!1)
[4]   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? )
Found substitution:
pref gets pref!1,
flt gets flt!1,
db gets db!1,
Instantiating quantified variables,
this simplifies to:
MAe.2 :

{-1}   NOT pref_filled(db!1, flt!1, pref!1)
          IMPLIES seat_exists(aircraft(flt!1), Next_seat(db!1, flt!1, pref!1))
[-2]   flt!1 = flt!2
[-3]   (# seat := Next_seat(db!1, flt!1, pref!1), pass := pas!1 #) = a!1
  |-------
[1]   pref_filled(db!1, flt!1, pref!1)
[2]   pass_on_flight(pas!1, flt!1, db!1)
[3]   db!1(flt!2)(a!1)
[4]   seat_exists(aircraft(flt!2), seat(a!1))
```

We now issue a GROUND command:

```
Rule? (ground)

Applying propositional simplification and decision procedures,

This completes the proof of MAe.2.

Q.E.D.


Run time  = 8.55 secs.
Real time = 1878.99 secs.

Wrote proof file /airlab/home/rwb/fm/pvs/tutorial-reservation-sys/pvs2/ops.prf
```

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:

```
(""
 (SKOSIMP*)
 (EXPAND "existence")
 (EXPAND "Make_assn")
 (AUTO-REWRITE "member")
 (SKOSIMP*)
 (INST?)
 (ASSERT)
```

```
(SPLIT -1)
(("1" (GROUND))
 ("2"
  (EXPAND "add")
  (FLATTEN)
  (LIFT-IF)
  (GROUND)
  (LEMMA "Next_seat_ax")
  (INST?)
  (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 (db: flt_db, flt: flight, pas: passenger, pref: preference):
         uniqueness(db) IMPLIES uniqueness(Make_assn(flt, pas, pref, db)))
```

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

```
Rule? (SKOSIMP*)
Repeatedly Skolemizing and flattening,
this simplifies to:
MAu :

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

We now expand uniqueness:

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

{-1}    (FORALL (a: [# pass: passenger, seat: [row, position] #]),
        (b: [# pass: passenger, seat: [row, position] #]), (flt: flight):
          member(a, db!1(flt)) AND member(b, db!1(flt)) AND pass(a) = pass(b)
              IMPLIES a = b)
  |-------
{1}   (FORALL (a: [# pass: passenger, seat: [row, position] #]),
        (b: [# pass: passenger, seat: [row, position] #]), (flt: flight):
          member(a, Make_assn(flt!1, pas!1, pref!1, db!1)(flt))
```

```
            AND member(b, Make_assn(flt!1, pas!1, pref!1, db!1)(flt))
              AND pass(a) = pass(b)
            IMPLIES a = b)
```

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

```
Rule?
(SKOSIMP*)
Repeatedly Skolemizing and flattening,
this simplifies to:
MAu :

[-1]   (FORALL (a: [# pass: passenger, seat: [row, position] #]),
       (b: [# pass: passenger, seat: [row, position] #]), (flt: flight):
         member(a, db!1(flt)) AND member(b, db!1(flt)) AND pass(a) = pass(b)
             IMPLIES a = b)
{-2}   member(a!1, Make_assn(flt!1, pas!1, pref!1, db!1)(flt!2))
{-3}   member(b!1, Make_assn(flt!1, pas!1, pref!1, db!1)(flt!2))
{-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. We know what they are, so we do it the non-automatic way this time:

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

We are ready for member to be rewritten so we expand it:

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

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

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

We would like to get rid of the WITH statements, so we issue a LIFT-IF command:

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

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

```
      ELSE IF pref_filled(db!1, flt!1, pref!1)
                OR pass_on_flight(pas!1, flt!1, db!1) THEN db!1(flt!2)(b!1)
        ELSE db!1(flt!2)(b!1)
        ENDIF
      ENDIF
[-4]   pass(a!1) = pass(b!1)
  |-------
[1]   a!1 = b!1
```

There are three IF expressions involving `pass_on_flight`. We decide to case split on this common expression to simplify the sequent into two smaller subgoals using the `CASE` command (TAB c):

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

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

```
[-5]   pass(a!1) = pass(b!1)
  |-------
[1]   a!1 = b!1
```

We issue an ASSERT to clean-up after the case-split:

```
Rule? (ASSERT)
Simplifying, rewriting, and recording with decision procedures,
this simplifies to:
MAu.1 :

[-1]   pass_on_flight(pas!1, flt!1, db!1)
{-2}   db!1(flt!2)(a!1)
{-3}   db!1(flt!2)(b!1)
[-4]   pass(a!1) = pass(b!1)
  |-------
{1}   db!1(flt!2)(a!1) AND db!1(flt!2)(b!1)
[2]   a!1 = b!1
```

This looks trivial so we issue another ASSERT:

```
Rule? (ASSERT)
Simplifying, rewriting, and recording with decision procedures,

This completes the proof of MAu.1.

MAu.2 :

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

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

The prover turns our attention to the second subgoal having finished the first. We issue an `ASSERT` (TAB a) here to simplify the sequent:

```
Rule? (ASSERT)
Simplifying, rewriting, and recording with decision procedures,
this simplifies to:
MAu.2 :

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

We are happy with the simplification achieved. We set our attention on formula `[1]` and expand `pass_on_flight`:

```
Rule? (EXPAND "pass_on_flight")
Expanding the definition of pass_on_flight,
this simplifies to:
MAu.2 :

[-1]   IF flt!1 = flt!2
        THEN IF pref_filled(db!1, flt!1, pref!1)
          THEN db!1(flt!2)(a!1)
        ELSE
          add((# seat := Next_seat(db!1, flt!1, pref!1), pass := pas!1 #),
              db!1(flt!1))(a!1)
        ENDIF
      ELSE db!1(flt!2)(a!1)
      ENDIF
[-2]   IF flt!1 = flt!2
        THEN IF pref_filled(db!1, flt!1, pref!1)
          THEN db!1(flt!2)(b!1)
        ELSE
          add((# seat := Next_seat(db!1, flt!1, pref!1), pass := pas!1 #),
              db!1(flt!1))(b!1)
        ENDIF
      ELSE db!1(flt!2)(b!1)
      ENDIF
[-3]   pass(a!1) = pass(b!1)
  |-------
{1}   (EXISTS (a: [# pass: passenger, seat: [row, position] #]):
         pass(a) = pas!1 AND member(a, db!1(flt!1)))
[2]   db!1(flt!2)(a!1) AND db!1(flt!2)(b!1)
[3]   a!1 = b!1
```

We are ready to instantiate formula `{1}`, but then realize that we are going to need two instances of it, one for `a!1` and one for `b!1`[14]. 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 1 "a!1")
Instantiating (with copying) the top quantifier in 1 with the terms:
a!1,
this simplifies to:
MAu.2 :

[-1]   IF flt!1 = flt!2
        THEN IF pref_filled(db!1, flt!1, pref!1)
          THEN db!1(flt!2)(a!1)
        ELSE
          add((# seat := Next_seat(db!1, flt!1, pref!1), pass := pas!1 #),
              db!1(flt!1))(a!1)
```

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

```
    ENDIF
  ELSE db!1(flt!2)(a!1)
  ENDIF
[-2]  IF flt!1 = flt!2
    THEN IF pref_filled(db!1, flt!1, pref!1)
      THEN db!1(flt!2)(b!1)
    ELSE
      add((# seat := Next_seat(db!1, flt!1, pref!1), pass := pas!1 #),
          db!1(flt!1))(b!1)
    ENDIF
  ELSE db!1(flt!2)(b!1)
  ENDIF
[-3]  pass(a!1) = pass(b!1)
  |-------
[1]   (EXISTS (a: [# pass: passenger, seat: [row, position] #]):
         pass(a) = pas!1 AND member(a, db!1(flt!1)))
{2}   pass(a!1) = pas!1 AND member(a!1, db!1(flt!1))
[3]   db!1(flt!2)(a!1) AND db!1(flt!2)(b!1)
[4]   a!1 = b!1
```

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

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

[-1]  IF flt!1 = flt!2
    THEN IF pref_filled(db!1, flt!1, pref!1)
      THEN db!1(flt!2)(a!1)
    ELSE
      add((# seat := Next_seat(db!1, flt!1, pref!1), pass := pas!1 #),
          db!1(flt!1))(a!1)
    ENDIF
  ELSE db!1(flt!2)(a!1)
  ENDIF
[-2]  IF flt!1 = flt!2
    THEN IF pref_filled(db!1, flt!1, pref!1)
      THEN db!1(flt!2)(b!1)
    ELSE
      add((# seat := Next_seat(db!1, flt!1, pref!1), pass := pas!1 #),
          db!1(flt!1))(b!1)
    ENDIF
  ELSE db!1(flt!2)(b!1)
  ENDIF
```

```
[-3]   pass(a!1) = pass(b!1)
  |-------
{1}   pass(b!1) = pas!1 AND member(b!1, db!1(flt!1))
[2]   pass(a!1) = pas!1 AND member(a!1, db!1(flt!1))
[3]   db!1(flt!2)(a!1) AND db!1(flt!2)(b!1)
[4]   a!1 = b!1
```

We expand add and member:

```
Rule? (EXPAND "add")
Expanding the definition of add,
this simplifies to:
MAu.2 :

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

Rule? (EXPAND "member")
Expanding the definition of member,
this simplifies to:
MAu.2 :

{-1}   IF flt!1 = flt!2
        THEN IF pref_filled(db!1, flt!1, pref!1)
          THEN db!1(flt!2)(a!1)
        ELSE (# seat := Next_seat(db!1, flt!1, pref!1), pass := pas!1 #) = a!1
              OR db!1(flt!1)(a!1)
```

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

Now we issue a GROUND command:

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

This completes the proof of MAu.2.

Q.E.D.


Run time  = 19.33 secs.
Real time = 1141.86 secs.

Wrote proof file /airlab/home/rwb/fm/pvs/tutorial-reservation-sys/pvs2/ops.prf
```

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

```
(""
 (SKOSIMP*)
 (EXPAND "uniqueness")
 (SKOSIMP*)
 (INST -1 "a!1" "b!1" "flt!2")
 (EXPAND "Make_assn")
 (EXPAND "member")
 (LIFT-IF)
 (CASE "pass_on_flight(pas!1, flt!1, db!1)")
 (("1" (ASSERT) (ASSERT))
  ("2"
   (ASSERT)
```

```
(EXPAND "pass_on_flight")
(INST-CP 1 "a!1")
(INST 1 "b!1")
(EXPAND "add")
(EXPAND "member")
(GROUND))))
```

This completes the two lemmas.

3.2.3 Proof of Theorem

We now must show that these two lemmas imply `Make_assn_inv`. We issue a `M-x pr` and issue the usual `SKOSIMP*` command (TAB *) to obtain:

```
Make_assn_inv :

{-1}   db_invariant(db!1)
  |-------
{1}   db_invariant(Make_assn(flt!1, pas!1, pref!1, db!1))
```

We expand `db_invariant` and flatten the sequent:

```
Rule? (EXPAND "db_invariant")
Expanding the definition of db_invariant,
this simplifies to:
Make_assn_inv :

{-1}   existence(db!1) AND uniqueness(db!1)
  |-------
{1}   existence(Make_assn(flt!1, pas!1, pref!1, db!1))
          AND uniqueness(Make_assn(flt!1, pas!1, pref!1, db!1))

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

{-1}   existence(db!1)
{-2}   uniqueness(db!1)
  |-------
[1]   existence(Make_assn(flt!1, pas!1, pref!1, db!1))
          AND uniqueness(Make_assn(flt!1, pas!1, pref!1, db!1))
```

Rather than use `(LEMMA "MAe") (INST? )` as before, we will illustrate the use of `(REWRITE "MAe")`:

```
Rule? (REWRITE "MAe")
Found matching substitution:
db gets db!1,
```

```
pref gets pref!1,
pas gets pas!1,
flt gets flt!1,
Rewriting using MAe,
this simplifies to:
Make_assn_inv :

[-1]   existence(db!1)
[-2]   uniqueness(db!1)
  |-------
{1}   TRUE AND uniqueness(Make_assn(flt!1, pas!1, pref!1, db!1))
```

The REWRITE command (TAB R) has matched the `existence(Make_assn(flt!1, pas!1, pref!1, db!1))` term and rewritten it to TRUE. We repeat the process with lemma MAu:

```
 (rewrite "MAu")


Found matching substitution:
db gets db!1,
pref gets pref!1,
pas gets pas!1,
flt gets flt!1,
Rewriting using MAu,
this simplifies to:
Make_assn_inv :

[-1]   existence(db!1)
[-2]   uniqueness(db!1)
  |-------
{1}   TRUE AND TRUE
```

which is obviously true so we issue ASSERT (TAB a):

```
Rule? (assert)

Simplifying, rewriting, and recording with decision procedures,
Q.E.D.


Run time  = 5.01 secs.
Real time = 662.10 secs.

Wrote proof file /airlab/home/rwb/fm/pvs/tutorial-reservation-sys/pvs2/ops.prf
NIL
>
```

The complete proof is:

```
(""
 (SKOSIMP*)
 (EXPAND "db_invariant")
 (FLATTEN)
 (REWRITE "MAe")
 (REWRITE "MAu")
 (ASSERT))
```

It is generally preferable to use `REWRITE` rather than `(LEMMA) (INST)` whenever it succeeds.

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 db_invariant(initial_state)
```

This theorem is easy to prove. In fact, the PVS strategy for proving TCCs 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 using the `GRIND` command (TAB G):

```
initial_state_inv :

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

Rule? (GRIND)
initial_state rewrites initial_state(flt)
  to emptyset[seat_assignment]
emptyset rewrites emptyset[seat_assignment](a)
  to FALSE
member rewrites member(a, emptyset[seat_assignment])
  to FALSE
existence rewrites existence(initial_state)
  to TRUE
initial_state rewrites initial_state(flt)
  to emptyset[seat_assignment]
emptyset rewrites emptyset[seat_assignment](a)
  to FALSE
member rewrites member(a, emptyset[seat_assignment])
  to FALSE
emptyset rewrites emptyset[seat_assignment](b)
  to FALSE
member rewrites member(b, emptyset[seat_assignment])
  to FALSE
uniqueness rewrites uniqueness(initial_state)
  to TRUE
```

```
db_invariant rewrites db_invariant(initial_state)
  to TRUE
Trying repeated skolemization, instantiation, and if-lifting,
Q.E.D.


Run time  = 1.36 secs.
Real time = 7.43 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(db)
    IMPLIES one_per_seat(Cancel_assn(flt,pas,db))


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

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,db(flt)) IMPLIES
                               seat(a) /= Next_seat(db,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 db, 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,db) =>
                 Cancel_assn(flt,pas,Make_assn(flt,pas,pref,db)) = db
```

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 (db: flt_db, flt: flight, pas: passenger, pref: preference):
         NOT pass_on_flight(pas, flt, db)
             => Cancel_assn(flt, pas, Make_assn(flt, pas, pref, db)) = db)
```

As always we skolemize with the `SKOSIMP*` command (TAB *):

```
Rule? (SKOSIMP*)
Repeatedly Skolemizing and flattening,
this simplifies to:
Make_Cancel :

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

We see that we must show that the database after modification is the same as it was before modification, whenever `NOT pass_on_flight(pas!1, flt!1, db!1)`[15]. But the database is a function, so we must prove the equivalence of two functions. Whenever this is necessary the `APPLY_EXTENSIONALITY` (TAB E) command must be used:

```
Rule?
(APPLY-EXTENSIONALITY 2)
Applying extensionality,
this simplifies to:
Make_Cancel :

  |-------
{1}   Cancel_assn(flt!1, pas!1, Make_assn(flt!1, pas!1, pref!1, db!1))(x!1)
          = db!1(x!1)
[2]   pass_on_flight(pas!1, flt!1, db!1)
[3]   Cancel_assn(flt!1, pas!1, Make_assn(flt!1, pas!1, pref!1, db!1)) = db!1
```

Notice that the new formula `{1}` is very similar to the old formula which has now become `[3]`. The idea here is that if we can show that $f_1(x) = f_2(x)$ for all values of `x` in its domain, then we know that $f_1 = f_2$. We won't need the old formula any more so we hide it with the `HIDE` command (TAB h):

```
Rule? (HIDE 3)
Hiding formulas:  3,
this simplifies to:
```

[15]One can move a formula from one side of the sequent to the other by adding a `NOT`. Notice that this is precisely what the system has done.

```
Make_Cancel :

  |-------
[1]   Cancel_assn(flt!1, pas!1, Make_assn(flt!1, pas!1, pref!1, db!1))(x!1)
          = db!1(x!1)
[2]   pass_on_flight(pas!1, flt!1, db!1)
```

Now from our previous experience, we remember that these functions are defined in terms of `add` and `member` that we were forced to constantly expand. To make these expansions happen automatically when we issue an `ASSERT` or `GROUND` command, we issue `AUTO-REWRITE` commands (TAB A) for them:

```
Rule? (AUTO-REWRITE "member")
Installing rewrite rule member
Installing rewrite rule member
Installing automatic rewrites:
  member
this simplifies to:
Make_Cancel :

  |-------
[1]   Cancel_assn(flt!1, pas!1, Make_assn(flt!1, pas!1, pref!1, db!1))(x!1)
          = db!1(x!1)
[2]   pass_on_flight(pas!1, flt!1, db!1)

Rule? (AUTO-REWRITE "add")
Installing rewrite rule add
Installing rewrite rule add
Installing automatic rewrites:
  add
this simplifies to:
Make_Cancel :

  |-------
[1]   Cancel_assn(flt!1, pas!1, Make_assn(flt!1, pas!1, pref!1, db!1))(x!1)
          = db!1(x!1)
[2]   pass_on_flight(pas!1, flt!1, db!1)
```

We expand `Cancel_assn` and `Make_assn`:

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

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

```
          WITH [(flt!1) :=
                  {a:
                    [# pass: passenger,
                       seat: [row, position] #]
                  |
                  member(a,
                         Make_assn(flt!1, pas!1,
                                   pref!1, db!1)(flt!1))
                      AND pass(a) /= pas!1}](x!1)
            = db!1(x!1)
[2]   pass_on_flight(pas!1, flt!1, db!1)

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

  |-------
{1}   IF pref_filled(db!1, flt!1, pref!1) THEN db!1
      ELSE db!1
          WITH [(flt!1) :=
                  add((# seat :=
                           Next_seat(db!1,
                                     flt!1, pref!1),
                         pass := pas!1 #),
                      db!1(flt!1))]
      ENDIF
        WITH [(flt!1) :=
                {a:
                  [# pass: passenger,
                     seat: [row, position] #]
                |
                member(a,
                       IF pref_filled(db!1, flt!1, pref!1)
                         THEN db!1(flt!1)
                       ELSE
                         add((# seat :=
                                  Next_seat(db!1,
                                            flt!1, pref!1),
                                pass := pas!1 #),
                             db!1(flt!1))
                       ENDIF)
                    AND pass(a) /= pas!1}](x!1)
          = db!1(x!1)
[2]   pass_on_flight(pas!1, flt!1, db!1)
```

To prepare for a GROUND command we issue a LIFT-IF command (TAB l):

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

  |-------
{1}   IF flt!1 = x!1
        THEN IF pref_filled(db!1, flt!1, pref!1)
          THEN {a: [# pass: passenger, seat: [row, position] #]
          | member(a, db!1(flt!1)) AND pass(a) /= pas!1}
              = db!1(x!1)
        ELSE {a: [# pass: passenger, seat: [row, position] #] |
          member(a,
                 add((# seat := Next_seat(db!1, flt!1, pref!1),
                        pass := pas!1 #),
                     db!1(flt!1)))
              AND pass(a) /= pas!1}
              = db!1(x!1)
        ENDIF
      ELSE IF pref_filled(db!1, flt!1, pref!1) THEN db!1(x!1) = db!1(x!1)
        ELSE IF flt!1 = x!1
            THEN {a: [# pass: passenger, seat: [row, position] #] |
            member(a,
                   add((# seat := Next_seat(db!1, flt!1, pref!1),
                          pass := pas!1 #),
                       db!1(flt!1)))
                AND pass(a) /= pas!1}
                = db!1(x!1)
          ELSE db!1(x!1) = db!1(x!1)
          ENDIF
        ENDIF
      ENDIF
[2]   pass_on_flight(pas!1, flt!1, db!1)
```

We now issue the `GROUND` (TAB g) command:

```
Rule? (GROUND)
member rewrites member(a, db!1(flt!1))
  to db!1(flt!1)(a)
member rewrites member(a, db!1(flt!1))
  to db!1(flt!1)(a)
add rewrites
  add((# seat := Next_seat(db!1, flt!1, pref!1), pass := pas!1 #),
      db!1(flt!1))(a)
  to (# seat := Next_seat(db!1, flt!1, pref!1), pass := pas!1 #) = a
          OR db!1(flt!1)(a)
```

```
member rewrites
  member(a,
         add((# seat := Next_seat(db!1, flt!1, pref!1), pass := pas!1 #),
             db!1(flt!1)))
  to (# seat := Next_seat(db!1, flt!1, pref!1), pass := pas!1 #) = a
          OR db!1(flt!1)(a)
Applying propositional simplification and decision procedures,
this yields  2 subgoals:
Make_Cancel.1 :

{-1}   pref_filled(db!1, flt!1, pref!1)
{-2}   flt!1 = x!1
  |-------
{1}   {a: [# pass: passenger, seat: [row, position] #]
      | db!1(flt!1)(a) AND pass(a) /= pas!1}
          = db!1(x!1)
[2]   pass_on_flight(pas!1, flt!1, db!1)
```

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 (TAB E) to prove two functions equal:

```
Rule? (APPLY-EXTENSIONALITY 1)
Applying extensionality,
this simplifies to:
Make_Cancel.1 :

[-1]   pref_filled(db!1, flt!1, pref!1)
[-2]   flt!1 = x!1
  |-------
{1}   (db!1(flt!1)(x!2) AND pass(x!2) /= pas!1) = db!1(x!1)(x!2)
[2]   {a: [# pass: passenger, seat: [row, position] #]
      | db!1(flt!1)(a) AND pass(a) /= pas!1}
          = db!1(x!1)
[3]   pass_on_flight(pas!1, flt!1, db!1)
```

We hide formula 2:

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

[-1]   pref_filled(db!1, flt!1, pref!1)
[-2]   flt!1 = x!1
  |-------
```

```
[1]   (db!1(flt!1)(x!2) AND pass(x!2) /= pas!1) = db!1(x!1)(x!2)
[2]   pass_on_flight(pas!1, flt!1, db!1)
```

We expand pass_on_flight:

```
Rule? (EXPAND "pass_on_flight")
Expanding the definition of pass_on_flight,
this simplifies to:
Make_Cancel.1 :

[-1]   pref_filled(db!1, flt!1, pref!1)
[-2]   flt!1 = x!1
  |-------
[1]   (db!1(flt!1)(x!2) AND pass(x!2) /= pas!1) = db!1(x!1)(x!2)
{2}   (EXISTS (a: [# pass: passenger, seat: [row, position] #]):
         pass(a) = pas!1 AND member(a, db!1(flt!1)))
```

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]   pref_filled(db!1, flt!1, pref!1)
[-2]   flt!1 = x!1
  |-------
[1]   (db!1(flt!1)(x!2) AND pass(x!2) /= pas!1) = db!1(x!1)(x!2)
{2}   pass(x!2) = pas!1 AND member(x!2, db!1(flt!1))
```

We now finish off this sequent with the GROUND command (TAB g):

```
Rule? (GROUND)
member rewrites member(x!2, db!1(flt!1))
  to db!1(flt!1)(x!2)
Applying propositional simplification and decision procedures,

This completes the proof of Make_Cancel.1.

Make_Cancel.2 :

{-1}   flt!1 = x!1
  |-------
{1}   pref_filled(db!1, flt!1, pref!1)
{2}   {a: [# pass: passenger, seat: [row, position] #] |
```

```
        ((# seat := Next_seat(db!1, flt!1, pref!1), pass := pas!1 #) = a
            OR db!1(flt!1)(a))
           AND pass(a) /= pas!1}
           = db!1(x!1)
[3]     pass_on_flight(pas!1, flt!1, db!1)
```

The prover now turns our attention to `Make_Cancel.2`. We issue an `(REPLACE -1 + RL)` to make the sequent easier for us to read[16].

```
Rule? (REPLACE -1 + RL)
Replacing using formula -1,
this simplifies to:
Make_Cancel.2 :

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

After replacing with a formula it is usually not needed again, so we hide it to cut down on the clutter:

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

  |-------
[1]    pref_filled(db!1, flt!1, pref!1)
[2]    {a: [# pass: passenger, seat: [row, position] #] |
        ((# seat := Next_seat(db!1, flt!1, pref!1), pass := pas!1 #) = a
            OR db!1(flt!1)(a))
           AND pass(a) /= pas!1}
           = db!1(flt!1)
[3]    pass_on_flight(pas!1, flt!1, db!1)
```

We are faced with proving the equality of two sets (i.e. functions) again so we issue an `APPLY-EXTENSIONALITY` command (TAB E) and hide the old formula as usual:

[16]The PVS decision procedures are powerful enough to handle this even if the `REPLACE` command is omitted. Nevertheless, often the discovery of a proof is easier when a `REPLACE` command is used in situations such as these.

```
Rule? (APPLY-EXTENSIONALITY 2)
Applying extensionality,
this simplifies to:
Make_Cancel.2 :

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

Rule? (HIDE 3)
Hiding formulas:  3,
this simplifies to:
Make_Cancel.2 :

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

We expand `pass_on_flight`:

```
Rule? (EXPAND "pass_on_flight")
Expanding the definition of pass_on_flight,
this simplifies to:
Make_Cancel.2 :

  |-------
[1]   (((# seat := Next_seat(db!1, flt!1, pref!1), pass := pas!1 #) = x!2
           OR db!1(flt!1)(x!2))
          AND pass(x!2) /= pas!1)
         = db!1(flt!1)(x!2)
[2]   pref_filled(db!1, flt!1, pref!1)
{3}   (EXISTS (a: [# pass: passenger, seat: [row, position] #]):
         pass(a) = pas!1 AND member(a, db!1(flt!1)))
```

and instantiate it as before:

```
Rule? (INST 3 "x!2")
Instantiating the top quantifier in 3 with the terms:
 x!2,
this simplifies to:
Make_Cancel.2 :

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

We issue a GROUND command (TAB g):

```
Rule? (GROUND)
member rewrites member(x!2, db!1(flt!1))
  to db!1(flt!1)(x!2)
Applying propositional simplification and decision procedures,
this simplifies to:
Make_Cancel.2 :

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

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 [2] to an IFF. This is done using the IFF command (TAB F):

```
Rule? (IFF 2)
Converting top level boolean equality into IFF form,
Converting equality to IFF,
this simplifies to:
Make_Cancel.2 :

  |-------
[1]   db!1(flt!1)(x!2)
{2}   (# seat := Next_seat(db!1, flt!1, pref!1), pass := pas!1 #) = x!2
           AND pass(x!2) /= pas!1
           IFF FALSE
```

```
[3]   pref_filled(db!1, flt!1, pref!1)
```

Now we finish it off with a GROUND:

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

This completes the proof of Make_Cancel.2.

Q.E.D.


Run time  = 14.19 secs.
Real time = 2292.41 secs.

Wrote proof file /airlab/home/rwb/fm/pvs/tutorial-reservation-sys/pvs2/ops.prf
NIL
>
```

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

```
(""
 (SKOSIMP*)
 (APPLY-EXTENSIONALITY 2)
 (HIDE 3)
 (AUTO-REWRITE "member")
 (AUTO-REWRITE "add")
 (EXPAND "Cancel_assn")
 (EXPAND "Make_assn")
 (LIFT-IF)
 (GROUND)
 (("1"
   (APPLY-EXTENSIONALITY 1)
   (HIDE 2)
   (EXPAND "pass_on_flight")
   (INST 2 "x!2")
   (GROUND))
  ("2"
   (REPLACE -1 + RL)
   (HIDE -1)
   (APPLY-EXTENSIONALITY 2)
   (HIDE 3)
   (EXPAND "pass_on_flight")
   (INST 3 "x!2")
   (GROUND)
   (IFF 2)
   (GROUND))))
```

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

```
Proof summary for theory ops
   Cancel_db_inv...........................................proved - complete
   MAe.....................................................proved - complete
   MAu.....................................................proved - complete
   Make_db_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(db, flt, pref) =>
        (EXISTS (x: seat_assignment):
          member(x, Make_assn(flt, pas, pref, db)(flt)) AND pass(x) = pas)

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

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

```
Lookup(flt: flight, pas: passenger, db: flt_db): [row,position] =
        seat(choose( {a | member(a,db(flt)) AND pass(a) = pas}))
```

and prove

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