Developing and Proving Algorithms with PVS
(Part I)

César A. Muñoz

NASA's Langley Research Center
Cesar.A.Munoz@nasa.gov
http://shemesh.larc.nasa.gov/people/cam

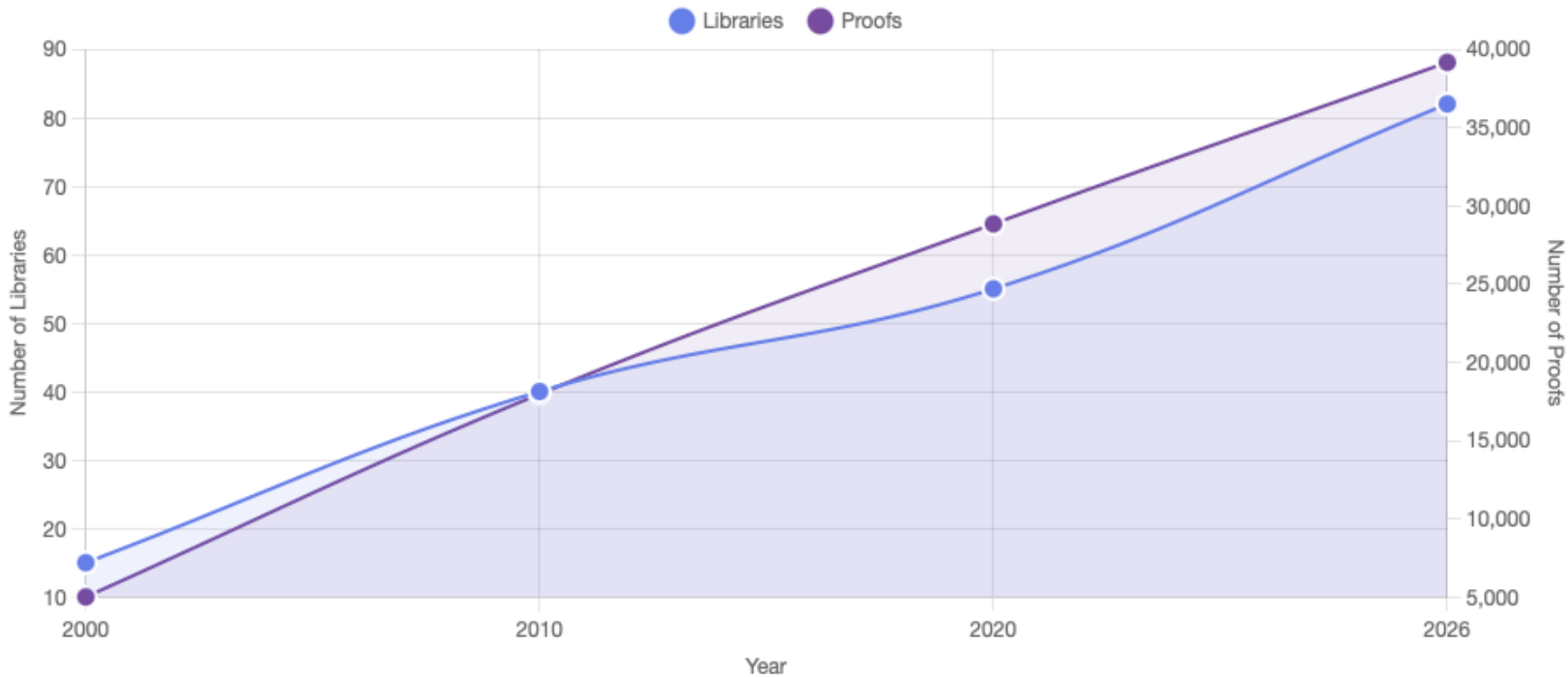
Fifteenth Summer School on Formal Techniques
May 23–29, 2026

The Prototype Verification System

- PVS is verification system developed by SRI International.[1]
- NASA LaRC has actively contributed to the development of PVS since 1992.[2]
  - NASALib: De-facto PVS library.
  - ProofLite: Batch proving utility.
  - PVSio: Animation of functional specifications via ground evaluation.
  - Plaidypvs: A framework for specifying and proving hybrid programs using differential dynamic logic.
  - pvsio_utils: PVSio libraries for parsing via PEG, reading/writing CSV, JSON, persistent key/value store, differential testing, ...
  - VSCode-PVS: A VSCode plugin for PVS.
  - Strategies for real-valued expressions, utilities for writing strategies, ...

[1] `https://pvs.csl.sri.com`

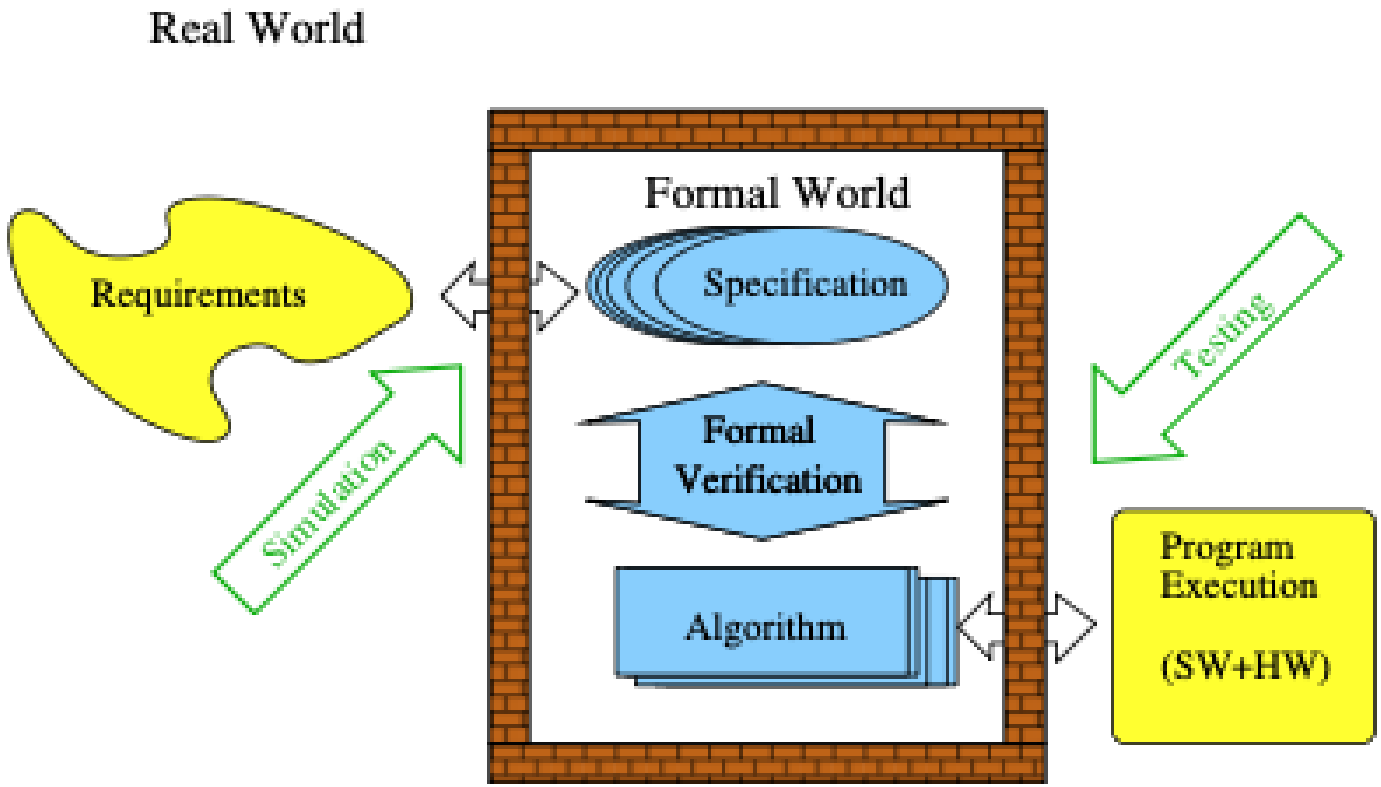
[2] `https://shemesh.larc.nasa.gov/fm/pvs`


NASALib: Libraries and Proofs Growth
Libraries
Proofs
Number of Libraries
Number of Proofs
Year
90
80
70
60
50
40
30
20
10
40,000
35,000
30,000
25,000
20,000
15,000
10,000
5,000
2000
2010
2020
2026

PVS

- Classical higher-order logic:
  `Shakespeare : THEOREM B OR NOT B`
- Strongly typed specification language.
  - Predicate subtyping
  - Dependent types
  - Inductive and co-inductive types
  - Parametric and prenex polymorphism
  - Overloading
- Built-in rational arithmetic:
  `BuiltinQArithmetic : THEOREM 3/3 = 1`
- Integrated decision procedures and automated proof-producing strategies: propositional logic, linear and non-linear arithmetic, finite state machines, equality with uninterpreted functions, ...

The World According to PVS



All Models Are Wrong

All models are wrong; the practical question is how wrong do they have to be to not be useful.

G. Box and N. Draper, Empirical Model Building and Response Surfaces, 1987.

Unmanned Aircraft System Traffic Management (UTM)

(PVS by Example)

- UTM is a highly automated, digital, distributed, and cooperative ecosystem designed to manage UAS flying at low altitudes.
- UTM relies on software systems exchanging data in real-time that provide services like
  - Geofencing.
  - Weather and wind integration.
  - Congestion management and collision avoidance.
  - Routing and flight planning.

Credits: D. Hart – NASA

UAS Service Supplier (USS)

(PVS by Example)

- A USS is a specific software platform within the UTM ecosystem that acts as the bridge between operators and the broader traffic management system.
- For a single, specific flight, an operator subscribes to a USS. The USS manages the operational volume, tracks aircraft telemetry, and provides additional services, e.g.
  - Flight planning and authorizations.
  - De-confliction.
  - Surveillance.

Credits: D. Hart – NASA

A USS in PVS[3]

(PVS by Example)

- USS will provide surveillance services for a set of aircraft within a given operational volume.
- Interface operations:
  - Create USS, including maximum capacity and operational volume (area & height).
  - Subscribe/unsubscribe aircraft operations.
  - Check if an aircraft is subscribed.
  - Get information about aircraft operation.
  - Check if maximum capacity has been reached.
  - Update aircraft state information (position and velocity).
  - Retrieve aircraft state.
  - Monitor traffic conflicts.
  - Monitor keep-in geofencing.

[3] **Disclaimer**: The specifications presented here are provided for pedagogical reasons. They do not represent a real implementation of a USS.

Theories

A PVS *theory* is a collection of declarations and definitions of mathematical objects.

```
% UAS Service Supplier
UTMO : THEORY
BEGIN

...

END UTMO
```

Parametric Theories

A *parametric* theory models a **family** of systems.

```
UTMO[AircraftOp:TYPE+,
     OperationalArea:TYPE+] : THEORY
BEGIN

...

END UTMO
```

Type Definitions

PVS is a **pure** functional language (it doesn't have a notion of state or memory)

```
% Type alias
AircraftId : TYPE = string

% Association list of tuples (id,info)
Operations : TYPE = list[[AircraftId,AircraftOp]]

% Record type
USS : TYPE = [#
  max_capacity : posnat,
  op_alt       : nnreal,
  op_area      : OperationalArea,
  operations   : Operations
#]
```

So far, is this model useful?

Type Definitions

PVS is a **pure** functional language (it doesn't have a notion of state or memory)

```
% Type alias
AircraftId : TYPE = string

% Association list of tuples (id,info)
Operations : TYPE = list[[AircraftId,AircraftOp]]

% Record type
USS : TYPE = [#
  max_capacity : posnat,
  op_alt       : nnreal,
  op_area      : OperationalArea,
  operations   : Operations
#]
```

So far, is this model useful?

Type Definitions

PVS is a **pure** functional language (it doesn't have a notion of state or memory)

```
% Type alias
AircraftId : TYPE = string

% Association list of tuples (id,info)
Operations : TYPE = list[[AircraftId,AircraftOp]]

% Record type
USS : TYPE = [#
  max_capacity : posnat,
  op_alt       : nnreal,
  op_area      : OperationalArea,
  operations   : Operations
#]
```

So far, is this model useful?

Type Definitions

PVS is a **pure** functional language (it doesn't have a notion of state or memory)

```
% Type alias
AircraftId : TYPE = string

% Association list of tuples (id,info)
Operations : TYPE = list[[AircraftId,AircraftOp]]

% Record type
USS : TYPE = [#
  max_capacity : posnat,
  op_alt       : nnreal,
  op_area      : OperationalArea,
  operations   : Operations
#]
```

So far, is this model useful?

Type Definitions

PVS is a **pure** functional language (it doesn't have a notion of state or memory)

```
% Type alias
AircraftId : TYPE = string

% Association list of tuples (id,info)
Operations : TYPE = list[[AircraftId,AircraftOp]]

% Record type
USS : TYPE = [#
  max_capacity : posnat,
  op_alt       : nnreal,
  op_area      : OperationalArea,
  operations   : Operations
#]
```

So far, is this model useful?

Create USS

```
mk_uss(capacity:posnat,
       alt:nnreal,
       area:OperationalArea) : USS = (#
   max_capacity := capacity,
   op_alt       := alt,
   op_area      := area,
   operations   := null
 #)
```

Subscribe Aircraft Operation

- Curried functions.
- Overwriting of records (and functions!).

```
subscribe(ac_id:AircraftId,op:AircraftOp)(uss:USS)
  : USS =
  IF length(uss`operations) < uss`max_capacity THEN
    uss WITH [
      `operations := cons((ac_id,op),uss`operations)
    ]
   ELSE
     uss
   ENDIF
```

Subscribe Aircraft Operation

- Curried functions.
- Overwriting of records (and functions!).

```
subscribe(ac_id:AircraftId,op:AircraftOp)(uss:USS)
  : USS =
  IF length(uss`operations) < uss`max_capacity THEN
    uss WITH [
      `operations := cons((ac_id,op),uss`operations)
    ]
   ELSE
     uss
   ENDIF
```

Unsubscribe Aircraft Operation

- Recursive functions.
- Termination measures
- Syntactic match for inductive data type and tuples

```
remove_op(ac_id:AircraftId,ops:Operations)
  : RECURSIVE Operations =
  CASES ops OF
    null : ops,
    cons(head,rest) :
      LET (id,op) = head IN
        IF id = ac_id THEN remove_op(ac_id,rest)
        ELSE cons(head,remove_op(ac_id,rest))
        ENDIF
  ENDCASES
  MEASURE length(ops)
```

Unsubscribe Aircraft Operation

- Recursive functions.
- Termination measures
- Syntactic match for inductive data type and tuples

```
remove_op(ac_id:AircraftId,ops:Operations)
  : RECURSIVE Operations =
  CASES ops OF
    null : ops,
    cons(head,rest) :
      LET (id,op) = head IN
        IF id = ac_id THEN remove_op(ac_id,rest)
        ELSE cons(head,remove_op(ac_id,rest))
        ENDIF
  ENDCASES
  MEASURE length(ops)
```

Unsubscribe Aircraft Operation

- Recursive functions.
- Termination measures
- Syntactic match for inductive data type and tuples

```
remove_op(ac_id:AircraftId,ops:Operations)
  : RECURSIVE Operations =
  CASES ops OF
    null : ops,
    cons(head,rest) :
      LET (id,op) = head IN
        IF id = ac_id THEN remove_op(ac_id,rest)
        ELSE cons(head,remove_op(ac_id,rest))
        ENDIF
  ENDCASES
  MEASURE length(ops)
```

Unsubscribe Aircraft Operation

- Recursive functions.
- Termination measures
- Syntactic match for inductive data type and tuples

```
remove_op(ac_id:AircraftId,ops:Operations)
  : RECURSIVE Operations =
  CASES ops OF
    null : ops,
    cons(head,rest) :
      LET (id,op) = head IN
        IF id = ac_id THEN remove_op(ac_id,rest)
        ELSE cons(head,remove_op(ac_id,rest))
        ENDIF
  ENDCASES
  MEASURE length(ops)
```

Termination Correctness Conditions

- Type-checker generates Type Correctness Conditions (TCCs) to be discharged by the user.
- However, most of these TCCs are automatically discharged by the prover.

```
% Termination TCC generated (at line 42, column 29) for
% remove_op(ac_id, rest)
remove_op_TCC1: OBLIGATION
  FORALL (ac_id: AircraftId, ops: Operations,
          head: [AircraftId,AircraftOp],
          rest: list[[AircraftId, AircraftOp]]):
    ops = cons(head, rest) IMPLIES
    FORALL (id: AircraftId, op: AircraftOp):
      (id = head`1 AND op = head`2) AND
      id = ac_id IMPLIES
      <<[[AircraftId, AircraftOp]](rest, ops)
```

Exercise 1

Using `remove_op`, define a function that unsubscribes an operation, e.g.,

```
unsubscribe(...) :  ... = ...
```

Checking if an Aircraft Operation Exists

- Inductive predicates are defined as the smallest fix point that satisfies the definition.

```
exists_op(ops:Operations)(ac_id:AircraftId)
  : INDUCTIVE bool =
  cons?(ops) AND
  LET (id,op) = car(ops) IN
    id = ac_id OR exists_op(cdr(ops))(ac_id)
```

- A measure is not needed, but to be well-defined they have to satisfy a syntactic positivity condition.
- If `P` is a predicate, i.e., `P` has the type `[T->bool]`, `P` is a PVS set, and `(P)` is the PVS type of elements in `P`. Therefore, `exists_op(ops)` is the set of aircraft identifiers in `ops`.

(On Functional Sub-Typing)

- In PVS, the type `[A1->B1]` is a sub-type of `[A2->B2]` iff `A1` = `A2` and `B1` is a sub-type of `B2`.
- The sub-typing is neither variant nor co-variant.
- Note that `{x:posnat | odd?(x)}` is not equal to `{x:nat | odd?(x)}` (they are not even comparable!):
  - `{x:posnat | odd?(x)}` has type `[posnat->bool]`.
  - `{x:nat | odd?(x)}` has type `[nat->bool]`.
  - `[posnat->bool]` is neither a sub-type nor a super-type of `[nat->bool]`.
- Tip: Always define sets using a common super-type at the base, so they are comparable!

(On Functional Sub-Typing)

- In PVS, the type `[A1->B1]` is a sub-type of `[A2->B2]` iff `A1` = `A2` and `B1` is a sub-type of `B2`.
- The sub-typing is neither variant nor co-variant.
- Note that `{x:posnat | odd?(x)}` is not equal to `{x:nat | odd?(x)}` (they are not even comparable!):
  - `{x:posnat | odd?(x)}` has type `[posnat->bool]`.
  - `{x:nat | odd?(x)}` has type `[nat->bool]`.
  - `[posnat->bool]` is neither a sub-type nor a super-type of `[nat->bool]`.
- Tip: Always define sets using a common super-type at the base, so they are comparable!

Types as Contracts

- Type judgements allow for pre-/post-condition specifications that restrict the original declaration without modifying it.
- Type judgements can be *manually* used as lemmas, but they are *automatically* used by the type-checker.

```
subscribe_j : JUDGEMENT
    subscribe(ac_id:AircraftId,op:AircraftOp)
     (uss:USS | length(uss`operations) < uss`max_capacity)
     HAS_TYPE
     {result:USS | exists_op(result`operations)(ac_id)}
```

Types as Contracts

- Type judgements allow for pre-/post-condition specifications that restrict the original declaration without modifying it.
- Type judgements can be *manually* used as lemmas, but they are *automatically* used by the type-checker.

```
subscribe_j : JUDGEMENT
    subscribe(ac_id:AircraftId,op:AircraftOp)
     (uss:USS | length(uss`operations) < uss`max_capacity)
     HAS_TYPE
     {result:USS | exists_op(result`operations)(ac_id)}
```

Types as Contracts

- Type judgements allow for pre-/post-condition specifications that restrict the original declaration without modifying it.
- Type judgements can be *manually* used as lemmas, but they are *automatically* used by the type-checker.

```
subscribe_j : JUDGEMENT
    subscribe(ac_id:AircraftId,op:AircraftOp)
     (uss:USS | length(uss`operations) < uss`max_capacity)
     HAS_TYPE
     {result:USS | exists_op(result`operations)(ac_id)}
```

Type Judgements Generate TCCs

```
subscribe_j :
  |---
{1} FORALL (ac_id: AircraftId, op: AircraftOp,
  uss: USS | length(uss`operations) < uss`max_capacity):
  exists_op(subscribe(ac_id, op)(uss)`operations)(ac_id)

Rule? (grind)

{-1} length(uss!1`operations) < uss!1`max_capacity
  |---
{1}  exists_op(cons((ac_id!1,op!1),uss!1`operations))(ac_id!1)

Rule? (expand "exists_op")

Q.E.D.
```

Type Judgements Generate TCCs

```
subscribe_j :
  |---
{1} FORALL (ac_id: AircraftId, op: AircraftOp,
  uss: USS | length(uss`operations) < uss`max_capacity):
  exists_op(subscribe(ac_id, op)(uss)`operations)(ac_id)

Rule? (grind)

{-1} length(uss!1`operations) < uss!1`max_capacity
 |---
{1}  exists_op(cons((ac_id!1,op!1),uss!1`operations))(ac_id!1)

Rule? (expand "exists_op")

Q.E.D.
```

Type Judgements For Recursive Definitions

```
remove_op_j : RECURSIVE JUDGEMENT
  remove_op(ac_id:AircraftId,ops:Operations)
  HAS_TYPE
  {result:Operations | NOT exists_op(result)(ac_id)}
```

- Often a recursive judgement is a better option than a regular judgement when specifying a type judgement of a recursive definition.
- TCCs are generated using an inductive principle that follows the recursive structure of the definition.

Inductionless Induction I

Base Case

```
remove_op_j_TCC1 :
    |---
{1}   FORALL (ac_id: AircraftId, ops: Operations,
  v: [d: [AircraftId, Operations] ->
       result: Operations | NOT exists_op(result)(d`1)]):
  (v = LAMBDA (ac_id: AircraftId, ops: Operations):
      remove_op(ac_id, ops))
      AND ops = null IMPLIES NOT exists_op(ops)(ac_id)

 Rule? (grind)
```

Inductionless Induction II

Base Case

```
{-1}   v!1 = ...
{-2}   null?(ops!1)
{-3}   exists_op(null)(ac_id!1)
    |---

Rule? (expand "exists_op")

Q.E.D.
```

Inductionless Induction I

Inductive Case

```
remove_op_j_TCC2 :
...

Rule? (skeep*)

{-1}   v = ...
{-2}   ops = cons(head, rest)
{-3}   id = head`1
{-4}   op = head`2
{-5}   exists_op(cons(head, v(ac_id, rest)))(ac_id)
  |---
{1}   id = ac_id

Rule? (typepred "v(ac_id,rest)")
```

Inductionless Induction II

Inductive Case

```
[-1]   v = ...
[-2]   ops = cons(head, rest)
[-3]   id = head‘1
[-4]   op = head‘2
[-5]   exists_op(cons(head, v(ac_id, rest)))(ac_id)
  |---
{1}   exists_op(v(ac_id, rest))(ac_id)
[2]   id = ac_id

Rule? (expand "exists_op" -5)
```

Inductionless Induction III

Inductive Case

```
[-1]   v = ...
[-2]   ops = cons(head, rest)
[-3]   id = head‘1
[-4]   op = head‘2
{-5}   head‘1 = ac_id OR exists_op(v(ac_id, rest))(ac_id)
  |---
[1]   exists_op(v(ac_id, rest))(ac_id)
[2]   id = ac_id

Rule? (assert)

Q.E.D.
```

Exercise 2

Define a type judgement for the function `unsbuscribe` stating that `result`, the output of the function, doesn't have an operation for `ac_id`, the input parameter of the function, i.e.,

```
unsubscribe_j : JUDGEMENT
  unsubscribe(...) HAS_TYPE ...
```

Prove the type judgement using `remove_op_j` as a lemma.

Cheat Sheet[4]

- `(assert ...)` : Apply decision procedures and auto-rewrites.
- `(grind ...)`: Super-duper strategy.
- `(ground ...)`: Apply propositional simplifications.
- `(flatten ...)` : Eliminate conjunctions in the antecedent and disjunctions in the consequent
- `(skeep ...)`: Eliminate `FORALL` quantifier in the consequent or `EXISTS` quantifier in the antecedent by introducing skolem variables.
- `(lemma ...)`: Introduce lemma to the consequent.
- `(expand ...)`: Expand definition.
- `(inst ...)`: Eliminate `FORALL` quantifier in the antecendent or `EXISTS` quantifier in consequent.

[4] See `https://pvs.csl.sri.com/doc/pvs-prover-guide.pdf` for a more comprehensive list of proof commands.

PVS Functions Are Total

Let's define a function that returns information about an aircraft operation.

```
get_info(ac_id:AircraftId)(uss:USS) : ...
```

- What is the return type?
- PVS functions are total, i.e., they are defined for every element in the domain.
  - Restrict the domain to `ac_id` that exists in `uss`.
  - Use a particular element of the domain to represent an exceptional output.
  - Return a Maybe/Option value. Constructors are `None` and `Some(<val>)`. Recognizers are `none?` and `some?`. Extractor is `val(<maybeobj>)`.

Getting Information About Aircraft Operation

```
IMPORTING structures@Maybe

find_op(ac_id:AircraftId,ops:Operations)
  : RECURSIVE Maybe[AircraftOp] =
  CASES ops OF
    null : None,
    cons(head,rest) :
      LET (id,op) = head IN
        IF id = ac_id THEN Some(op)
        ELSE find_op(ac_id,rest)
        ENDIF
  ENDCASES
  MEASURE length(ops)

get_info(ac_id:AircraftId)(uss:USS)
  : Maybe[AircraftOp] =
  find_op(ac_id,uss`operations)
```

Getting Information About Aircraft Operation

```
IMPORTING structures@Maybe

find_op(ac_id:AircraftId,ops:Operations)
  : RECURSIVE Maybe[AircraftOp] =
  CASES ops OF
    null : None,
    cons(head,rest) :
      LET (id,op) = head IN
        IF id = ac_id THEN Some(op)
        ELSE find_op(ac_id,rest)
        ENDIF
  ENDCASES
  MEASURE length(ops)

get_info(ac_id:AircraftId)(uss:USS)
  : Maybe[AircraftOp] =
  find_op(ac_id,uss`operations)
```

Getting Information About Aircraft Operation

```
IMPORTING structures@Maybe

find_op(ac_id:AircraftId,ops:Operations)
  : RECURSIVE Maybe[AircraftOp] =
  CASES ops OF
    null : None,
    cons(head,rest) :
      LET (id,op) = head IN
        IF id = ac_id THEN Some(op)
        ELSE find_op(ac_id,rest)
        ENDIF
  ENDCASES
  MEASURE length(ops)

get_info(ac_id:AircraftId)(uss:USS)
  : Maybe[AircraftOp] =
  find_op(ac_id,uss`operations)
```

Testing The Model (I)

- Define some uninterpreted constants.
- Define some syntactic sugar.
  - Infix operators can be defined using a pre-defined set of symbols (they have to be preceded by a semi-colon).
  - Macros are automatically expanded by the prover.

```
OpAlt  : nnreal
OpArea : OperationalArea
Ac0,Ac1,Ac2 : AircraftId
Op0,Op1,Op2 : AircraftOp

;++(uss:USS,f:[USS->USS]) : MACRO USS = f(uss)
```

Testing The Model (I)

- Define some uninterpreted constants.
- Define some syntactic sugar.
  - Infix operators can be defined using a pre-defined set of symbols (they have to be preceded by a semi-colon).
  - Macros are automatically expanded by the prover.

```
OpAlt  : nnreal
OpArea : OperationalArea
Ac0,Ac1,Ac2 : AircraftId
Op0,Op1,Op2 : AircraftOp

;++(uss:USS,f:[USS->USS]) : MACRO USS = f(uss)
```

Testing The Model (I)

- Define some uninterpreted constants.
- Define some syntactic sugar.
  - Infix operators can be defined using a pre-defined set of symbols (they have to be preceded by a semi-colon).
  - Macros are automatically expanded by the prover.

```
OpAlt  : nnreal
OpArea : OperationalArea
Ac0,Ac1,Ac2 : AircraftId
Op0,Op1,Op2 : AircraftOp

;++(uss:USS,f:[USS->USS]) : MACRO USS = f(uss)
```

Testing The Model (II)

- Define test objects.
- Define test lemmas.
- Prove them (usually with `(grind)`)

```
test_uss : USS =
    mk_uss(3,OpAlt,OpArea) ++
      subscribe(Ac0,Op0) ++ subscribe(Ac1,Op1) ++
      subscribe(Ac2,Op2) ++ unsubscribe(Ac0)

test_0 : LEMMA
    none?(get_info(Ac0)(test_uss))

...
  |---
{1}   none?(get_info(Ac0)(test_uss))

Rule? (grind)

Q.E.D.
```

Testing The Model (II)

- Define test objects.
- Define test lemmas.
- Prove them (usually with `(grind)`)

```
test_uss : USS =
    mk_uss(3,OpAlt,OpArea) ++
      subscribe(Ac0,Op0) ++ subscribe(Ac1,Op1) ++
      subscribe(Ac2,Op2) ++ unsubscribe(Ac0)

test_0 : LEMMA
    none?(get_info(Ac0)(test_uss))

...
  |---
{1}   none?(get_info(Ac0)(test_uss))

Rule? (grind)

Q.E.D.
```

Testing The Model (II)

- Define test objects.
- Define test lemmas.
- Prove them (usually with `(grind)`)

```
test_uss : USS =
    mk_uss(3,OpAlt,OpArea) ++
      subscribe(Ac0,Op0) ++ subscribe(Ac1,Op1) ++
      subscribe(Ac2,Op2) ++ unsubscribe(Ac0)

test_0 : LEMMA
    none?(get_info(Ac0)(test_uss))

...
  |---
{1}   none?(get_info(Ac0)(test_uss))

Rule? (grind)

Q.E.D.
```

Testing The Model (II)

```
test_1 : LEMMA
    get_info(Ac1)(test_uss) = Some(Op1)

...
  |---
{1}   get_info(Ac1)(test_uss) = Some(Op1)

Rule?  (grind)

test_1.1 :

{-1}   Ac1 = Ac0
  |---
...
```

What happens here?

Exercise 3

Fix `test_1` so that it holds.

Hint: What is known about `Ac0`, `Ac1`, and `Ac2`? What should be assumed?

Exercise 4

Note that one aircrtaft operation can saturate the USS, i.e.,

```
full_capacity(uss:USS) : bool =
  length(uss`operations) = uss`max_capacity

test_dos : USS =
  mk_uss(3,OpAlt,OpArea) ++ subscribe(Ac0,Op0) ++
    subscribe(Ac0,Op0) ++ subscribe(Ac0,Op0)

DoS : LEMMA
   full_capacity(test_dos)
```

Fix `subscribe` so that only one operation per aircraft can be subscribed. After the change, make sure all TCCs and LEMMAS are still valid.

Exercise 5

Define the function

```
update(ac_id:AircraftId,op:AircraftOp)(uss:USS)
  : USS = ...
```

that modifies the operation for aircraft `ac_id` to `op`. The function does nothing if the aircraft is not subscribed. Write a lemma to test the functionality.

Hint: First, define a recursive function on `Operations` that change an element in the list. Then, use this function to define `update`.