Software Analysis in Practice

Willard Rafnsson
Carnegie-Mellon University

Course: 18-732
Spring 2015
Topic Today: Software Analysis

0. Attacks
1. Software Security Architectures
2. Security Analysis of Software
3. Language-Based Security
4. Run-time Security Enforcement
Software Engineering, Holy Grail

we desire software that is
- correct,
- easy to use,
- hard to misuse,
- easy to develop,
- easy to maintain.

each in what degree: depends on scenario.
Software Engineering, Holy Grail

we desire software that is

- correct,  focus today
- easy to use,
- hard to misuse,
- easy to develop,
- easy to maintain.

affects other goals profoundly; correct software is easy to maintain, hard to misuse.
Software Engineering, Holy Grail

we desire software that is
- correct,
- easy to use,
- hard to misuse,
- easy to develop,
- easy to maintain.

correctness beyond any doubt: mathematics.
formal methods is born!

reason about programs:
- semantics
- specification
- tools
Formal Methods:

(hi)story time
Formal Methods:

Halting Problem

[Turing, 1936]
Formal Methods:

Halting Problem

“does $P$ halt given $i$?”

[Turing, 1936]
Halting Problem

Formal Methods: [Turing, 1936]

“does $P$ halt given $i$?”

true or false, for all $P$ and $i$.

---

```plaintext
Put_Line ("Hello, world!");

while True loop
    Put ("heya... ");
end;

while i != 0 loop
    Put ("i is not 0... ");
end;
```
Halting Problem

"does \( P \) halt given \( i \)?"

true or false, for all \( P \) and \( i \).
halts(\( P, i \)).
let's implement halts...

```
Put_Line ("Hello, world!");
```

```
while True loop
  Put ("heya... ");
end;
```

```
while i != 0 loop
  Put ("i is not 0... ");
end;
```
Formal Methods:

**Halting Problem**

"does $P$ halt given $i$?"

**Theorem:**

$halts(P, i)$ is undecidable.

no $P'$ computes $halts$.
(always eventually returns yes/no)


**Halting Problem**

"does $P$ halt given $i$?"

**Theorem:**

$\text{halts}(P, i)$ is undecidable.

(why: Gödel’s incompleteness result).

---

[Formal Methods:

- **Turing, 1936**
- **Gödel, 1931**

Program input: no $P'$ computes $\text{halts}$. (always eventually returns yes/no)
Theorem: \( \text{halts}(P, i) \) is undecidable. (why: Gödel’s incompleteness result).
Halting Problem

“does \( P \) halt given \( i \)?”

Theorem: \( \text{halts}(P, i) \) is undecidable.

Corollary: for any \( p \) that is not trivial, \( p(P) \) is undecidable.

Formal Methods:

[Turing, 1936]

[Rice, 1953]
Formal Methods:

**What can we do?**

no tool can prove whether or not any given program $P$ satisfies any given specification $p$. 

always eventually answer yes/no
What *can* we do?

**no tool** can prove whether or not any given program $P$ satisfies any given specification $p$.

**formal methods** explores
- allowing false alarms,
- imposing restrictions on $P$ or $p$, and
- requiring assistance from a human.
What can we do?

no tool can prove whether or not any given program P satisfies any given specification p.

formal methods:
- allowing false alarms,
- imposing restrictions on P or p, and
- requiring assistance from a human.

There are many ways to prove that a program is correct.

(model checking, types, ...)

today we focus on approaches which help a programmer write verifiable interface specifications: contracts.
Formal Methods:

Flowcharts

[Floyd, 1967]
Formal Methods:

Flowcharts

[Floyd, 1967]

R := 0;
while (R + 1)^2 <= N loop
  R := R + 1;
end loop;
Flowcharts

$R := 0$
$\text{while } (R + 1)^2 \leq N \text{ loop}$
$R := R + 1$
$\text{end loop}$

Formal Methods:

- state change
- branching

[Floyd, 1967]
Formal Methods:

**Flowcharts**

\[
R := 0; \\
\text{while } (R + 1)^2 \leq N \text{ loop} \\
R := R + 1; \\
\text{end loop;}
\]

[Floyd, 1967]
Formal Methods:

Flowcharts

\begin{verbatim}
R := 0;
while (R + 1)^2 <= N loop
  R := R + 1;
end loop;
\end{verbatim}

\[ R := 0; \]
\[ \text{while } (R + 1)^2 <= N \text{ loop} \]
\[ R := R + 1; \]
\[ \text{end loop;} \]

[\text{[Floyd,1967]}]
Formal Methods:

Flowcharts

```c
R := 0;
while (R + 1)^2 <= N loop
  R := R + 1;
end loop;
```

[Floyd, 1967]
Formal Methods:

**Flowcharts**

R := 0;
while \((R + 1)^2 \leq N\) loop
  R := R + 1;
end loop;

true whenever
condition is entered
(and condition does
not change state)
Proof that after loop, 
R = integer square root of N (if 0 ≤ N initially)
Formal Methods:

Hoare Triples

\[ \{ P \} S \{ Q \} \]

- predicate on state (precondition)
- program
- predicate on state (postcondition)

[Hoare, 1969]
Formal Methods:

**Hoare Triples**

$a$ Hoare triple: a *predicate*.

“If we start executing $S$ in state satisfying $P$, we end up in a state satisfying $Q$."

[Hoare, 1969]
Formal Methods:

**Hoare Triples**

\[
\{ P \} S \{ Q \}
\]

a Hoare triple: a predicate.

“if we start executing \( S \) in state satisfying \( P \), we end up up in a state satisfying \( Q \).”

```plaintext
R := 0;
while (R + 1)^2 <= N loop
    R := R + 1;
end loop;
```
Formal Methods:

**Hoare Triples**

\[
\{ P \} \; S \; \{ Q \} 
\]

a **Hoare triple**: a *predicate*.

“if we start executing \( S \) in state satisfying \( P \), we end up in a state satisfying \( Q \).”

---

```plaintext
{ 0 <= N }
R := 0;
{ R^2 <= N }
while (R + 1)^2 <= N loop
  { R^2 <= N and (R+1)^2 <= N }
  R := R + 1;
  { R^2 <= N }
end loop;
{ R^2 <= N and N < (R+1)^2 }
```
Hoare Triples

\{ P \} S \{ Q \}

a Hoare triple: a predicate.

“if we start executing $S$ in state satisfying $P$, we end up in a state satisfying $Q$.”

\[\begin{align*}
\{ 0 \leq N \} \\
R := 0; \\
\{ R^2 \leq N \} \\
\text{while } (R + 1)^2 \leq N \text{ loop} \\
\quad \{ R^2 \leq N \text{ and } (R+1)^2 \leq N \} \\
R := R + 1; \\
\quad \{ R^2 \leq N \} \\
\text{end loop;} \\
\{ R^2 \leq N \text{ and } N < (R+1)^2 \}
\end{align*}\]

R$^2$ $\leq$ N is a (loop) invariant;
- holds on loop entry,
- holds after each iteration
- by induction, will hold on loop exit.
Formal Methods:

Weakest Precondition [Dijkstra, 1976]

\{ P \} S \{ Q \}

predicate on state (precondition)

program

predicate on state (postcondition)
Weakest Precondition [Dijkstra, 1976]

\[
\{ P \} \; S \; \{ Q \} \\
\]

- \( sp(S,P) \) = strongest postcondition
  - \( Q \) for which \( \{ P \} \; S \; \{ Q \} \) holds.

- \( wp(S,Q) \) = weakest precondition
  - \( P \) for which \( \{ P \} \; S \; \{ Q \} \) holds.

If you know \( P \) holds in initial state, \( sp(S,P) \) is the strongest guarantee you get by running \( S \).

If you desire \( Q \) after running \( S \), \( wp(S,Q) \) is the weakest guarantee your initial state must satisfy.
Weakest Precondition [Dijkstra, 1976]

\{ P \} S \{ Q \}

- \textit{predicate on state (precondition)}
- \textit{program}
- \textit{predicate on state (postcondition)}

\( sp(S,P) \) = strongest postcondition

\( Q \) for which \( \{ P \} S \{ Q \} \) holds.

\( wp(S,Q) \) = weakest precondition

\( P \) for which \( \{ P \} S \{ Q \} \) holds.

**Theorem:** for every \( P, S, Q \):

if \( \{ P \} S \{ Q \} \) holds, then

\( sp(S,P) \Rightarrow Q \), and \( P \Rightarrow wp(S,Q). \)

**idea:** calculate the conditions! now we can build tools; a proof checker!

**infer** most conditions in example on previous slide
Formal Methods:

Contracts (Eiffel) [Meyer, 1988]

- Method **precondition**: caller’s responsibility to establish, implementation can assume on entry.
- Method **postcondition**: implementation’s responsibility to establish, caller can assume on return.

verification that is **modular**.
Formal Methods:

Fast-Forward
Formal Methods, Today

techniques **enabling tool building**
- symbolic execution
  ("execute" program for many values at a time)
- abstract interpretation
  (loops; $n=0$ or $n=1$ or … implied by $0 \leq n$)
- instantiating quantifiers

**tools built**
- Eiffel, Spec#, Dafny, KeY, ...

https://github.com/johnyf/tool_lists/blob/master/verification_synthesis.md
today:

SPARK
SPARK?

Related Work

[...] bla bla bla SPARK Ada [42] has a static data-flow analysis component which can enforce information-flow properties bla [...]
SPARK?

Related Work

[...] bla bla bla SPARK Ada [42] has a static data-flow analysis component which can enforce information-flow properties bla [...]
Ada (programming language)

Ada is a structured, statically typed, imperative, wide-spectrum, and object-oriented high-level computer programming language, extended from Pascal and other languages. [...] Ada was originally designed [...] under contract to the United States Department of Defense (DoD) from 1977 to 1983 to supersede the hundreds of programming languages then used by the DoD.
Ada (programming language)

From Wikipedia, the free encyclopedia

Ada is a structured, statically typed, imperative, wide-spectrum, and object-oriented high-level computer programming language, extended from Pascal and other languages. [...] Ada was originally designed [...] under contract to the United States Department of Defense (DoD) from 1977 to 1983 to supersede the hundreds of programming languages then used by the DoD.
Ada (programming language)

From Wikipedia, the free encyclopedia

**Ada** is a structured, statically typed, imperative, wide-spectrum, and object-oriented high-level computer programming language, extended from Pascal and other languages. [...] 

Ada was originally designed [...] under contract to the United States Department of Defense (DoD) from 1977 to 1983 to supersede the hundreds of programming languages then used by the DoD.

What groundbreaking developments/uses of Ada have occurred since?
Developed since 1983.
Much more than a simple data-flow analysis for Ada.
All the “cool kids” use it.
Completely redesigned last year.
"SPARK consists of
“**SPARK** consists of which, taken together, ensure that **ultra-low defect** software can be deployed in application domains where **high-reliability** must be assured.” (e.g. where **safety** and **security** are key requirements)
where safety and security are key requirements
SPARK consists of which, taken together, ensure that ultra-low defect software can be deployed in application domains where high-reliability must be assured. (e.g. where safety and security are key requirements)
I'M INTRIGUED.

TELL ME MORE.
SPARK:
Features

- Programming Language
- Design Method
- Verification Toolset
- High Reliability
SPARK:

Features
SPARK:

Features

**language**: subset of Ada; **no**

- pointers, dynamic data structures, concurrency (yet), controlled types (for compiler optimization), gotos, side-effects in expressions (incl. f(4)), exception handling.

Features

**Language:** subset of Ada; no

- pointers, dynamic data structures, concurrency (yet), controlled types (for compiler optimization), gotos, side-effects in expressions (incl. f(4)), exception handling.

SPARK:

Features

language: subset of Ada; no

- pointers, dynamic data structures, concurrency (yet), controlled types (for compiler optimization), gotos, side-effects in expressions (incl. f(4)), exception handling.


consequence: much easier to analyze programs.

- no heap!
- no \{buffer, integer\} overflows!

simpler control flow: knowing what a program does by “looking at” the source code
Features

**language:** subset of Ada; **no**
- pointers, dynamic data structures, concurrency (yet), controlled types (for compiler optimization), gotos, side-effects in expressions (incl. $f(4)$), exception handling.


consequence: much easier to analyze programs.

- no heap!
- no `{buffer, integer}` overflows!

simpler **control flow**; knowing what a program does by “looking at” the source code
Features

toolset: two components.

- flow analysis checks dependency annotations in program to ensure the dependencies (in variables, parameter and global) created by running a procedure are intentional.

```
program generate PDG analyze
```

program → generate → PDG → analyze

```
OK NOT OK
```

SPARK:
Features

**toolset:** two components.

- **flow analysis**
  checks **dependency** annotations in program to ensure the dependencies (in variables, parameter and global) created by running a procedure are intentional.

- **verification**
  contract annotation; **pre/post** conditions, **contract cases**.

  "GNATprove is a formal verification tool for Ada, based on the GNAT compiler, Why3 platform and Alt-Ergo prover."

  [webpage link](http://www.open-do.org/projects/hi-lite/gnatprove/)
does someone actually use SPARK?
Yes!

looking for career opportunities? pay close attention.
Applications, safety

- aviation, commercial
- traffic management
- medical
- space

Rolls-Royce Trent jet engines

Lockheed Martin C130J

ACAMS system
Applications, safety

- aviation, military
- traffic management
- medical
- space

EuroFighter Typhoon
(Eurofighter Jagdflugzeug GmbH)
(Airbus, BAE Systems, Alenia Aermacchi)

AerMacchi M346
(Alenia Aermacchi)

Harrier GR9
(BAE Systems)
Applications, safety

- aviation
- traffic management
- medical
- space

NATS iFACTS system (UK)
Applications, safety

- aviation
- traffic management
- medical
- space

LifeFlow ventricular assist device
(University of Virginia)

CubeSat
(Vermont Technical College)
Applications, security

- cross-domain guard
- smart card OS
- biometrics software
- multi-level workstation
- separation kernel

Turnstile, and SecureOne, by Rockwell Collins (for moving information between security domains,)
Applications, security

- cross-domain guard
- smart card OS
- biometrics software
- multi-level workstation
- separation kernel

MULTOS, open standard by MULTOS Consortium
Applications, security

- cross-domain guard
- smart card OS
- biometrics software
- multi-level workstation
- separation kernel

Tokeneer, by NSA
SPARK:

Applications, security

- cross-domain guard
- smart card OS
- biometrics software
- multi-level workstation
- separation kernel

by Secunet
SPARK:

Applications, security

- cross-domain guard
- smart card OS
- biometrics software
- multi-level workstation
- separation kernel

Muen, by secunet
Advantage: Its History

there are new good/better tools.
but SPARK has been used for decades.
- undergone review,
- satisfies standards,
- case studies, ...

industry trusts SPARK;
SPARK has its “foot in the door”.

SPARK:

How reliable is SPARK?

Ada has an operational **semantics** flow analysis & proofs **sound** for semantics,

Ada *compiler* is **not certified**;

its semantics != operational semantics.

giving such guarantees is hard

- OS, HW (cache, memory, …), concurrency, …

**Remember**: no verifier is perfect; each provides guarantees, based on **assumptions**.
let's play with SPARK.
Getting Started

Tools

- Ada 2012 compiler:
- SPARK 2014 prover:

Docs

- SPARK toolset user guide:
- SPARK reference manual:
- Ada reference manual (incl. standard library):
  http://www.adaic.org/resources/add_content/standards/12rm/html/RM-TTL.html
- More (websites, books, videos, tutorials, etc.):
  http://www.adacore.com/adaanswers/resources
  http://www.adacore.com/developers/documentation
SPARK:

Getting Started
Getting Started

compile:
  gnatmake package.adb

prove:
  gnatprove -P project.gpr

make sure your stack is big enough.
SPARK:

Getting Started

hello.adb

```ada
with Ada.Text_IO; use Ada.Text_IO;
procedure Hello is
begin
    Put_Line ("Hello World. Welcome to GNAT");
end Hello;
```
Getting Started

```ada
with Ada.Text_IO; use Ada.Text_IO;
procedure Hello is
begin
  Put_Line ("Hello World. Welcome to GNAT");
end Hello;
```

```shell
[willard@thor]$ ls
hello.adb
[willard@thor]$ gnatmake hello.adb
gcc -c hello.adb
gnatbind -x hello.ali
gnatlink hello.ali
[willard@thor]$ ls
hello hello.adb hello.ali hello.o
[willard@thor]$ ./hello
Hello World. Welcome to GNAT
[willard@thor]$ 
```
SPARK:

Getting Started

toolset:

Flow Analysis
Getting Started

toolset:

Flow Analysis

DEMO
SPARK:

Getting Started

toolset:

Verification (Contracts)
pkg.adb ("code")

package body Pkg is
    procedure Absolute
        (X : in Integer;
         Y : out Integer)
    is begin
        if X < 0
            then Y := -X;
        else Y := X;
        end if;
    end Absolute;
end Pkg;

pkg.ads ("interface")

package Pkg is
    procedure Absolute
        (X : in Integer;
         Y : out Integer);
end Pkg;

proj.gpr ("project")

project Proj is
    for Source_Dirs use (".");
end Proj;
pkg.adb ("code")

```
package body Pkg is
    procedure Absolute
        (X : in Integer;
         Y : out Integer)
    is begin
        if X < 0
            then Y := -X;
        else Y := X;
        end if;
    end Absolute;
end Pkg;
```

pkg.ads ("interface")

```
package Pkg is
    procedure Absolute
        (X : in Integer;
         Y : out Integer);
end Pkg;
```

shell

```
[willard@thor]$ gnatprove -P proj.gpr
Phase 1 of 3: frame condition computation ...
Phase 2 of 3: analysis and translation to intermediate language ...
Phase 3 of 3: generation and proof of VCs ...
warning: no bodies have been analyzed by GNATprove enable_analysis_of_a_body_using_SPARK_Mode
[willard@thor]$  
```
pragma SPARK_Mode (On);
package body Pkg is
    procedure Absolute
        (X : in Integer;
         Y : out Integer)
    is begin
        if X < 0
            then Y := -X;
        else Y := X;
        end if;
    end Absolute;
end Pkg;

pragma SPARK_Mode (On);
package Pkg is
    procedure Absolute
        (X : in Integer;
         Y : out Integer);
end Pkg;
pkg.adb

pragma SPARK_Mode (On);
package body Pkg is
  procedure Absolute
    (X : in Integer;
     Y : out Integer)
    is begin
      if X < 0
        then Y := -X;
      else Y := X;
      end if;
    end Absolute;
end Pkg;

pkg.ads

pragma SPARK_Mode (On);
package Pkg is
  procedure Absolute
    (X : in Integer;
     Y : out Integer);
end Pkg;

shell

[willard@thor]$ gnatprove -P proj.gpr
Phase 1 of 3: frame condition computation ...
Phase 2 of 3: analysis and translation to intermediate language ...
Phase 3 of 3: generation and proof of VCs ...
analyzing Pkg, 0 checks
analyzing Pkg.Absolute, 1 checks
pkg.adb:8:17: warning: overflow check might fail
gnatprove: *** compilation phase failed
gprbuild: ** error during generation and proof of VCs, aborting.
[willard@thor]$
fix with a **precondition**; anyone calling Pkg must show that \( X \neq \text{Integer}'\text{First} \)
pkg.adb

```
pragma SPARK_Mode (On);
package body Pkg is
    procedure Absolute
        (X : in Integer;
         Y : out Integer)
         is begin
            if X < 0
            then Y := -X;
            else Y := X;
            end if;
        end Absolute;
end Pkg;
```

gpkg.ads

```
pragma SPARK_Mode (On);
package Pkg is
    procedure Absolute
        (X : in Integer;
         Y : out Integer)
        with Pre => (X /= Integer'First);
end Pkg;
```

fix with a \texttt{precondition}; anyone calling Pkg must show that \(X \neq \text{Integer'}\text{First}\).

what’s a reasonable precondition we can add?

shell

```
$ gnatprove -P proj.gpr
Phase 1 of 3: frame condition computation ...
Phase 2 of 3: analysis and translation to intermediate language ...
Phase 3 of 3: generation and proof of VCs ...
analyzing Pkg, 0 checks
analyzing Pkg.Absolute, 1 checks
Summary logged in /home/willard/Documents/spark-lecture/abs0/abs11/gnatprove/gnatprove.out
```

[willard@thor]$
pkg.adb

pragma SPARK_Mode (On);
package body Pkg is
    procedure Absolute
        (X : in Integer;
         Y : out Integer)
    is begin
        if X < 0 then Y := -X;
        else Y := X;
    end if;
    end Absolute;
end Pkg;

dpkg.ads

pragma SPARK_Mode (On);
package Pkg is
    procedure Absolute
        (X : in Integer;
         Y : out Integer)
    with Pre  => (X /= Integer'First),
         Post => (0 <= Y);
end Pkg;
pkg.adb

pragma SPARK_Mode (On);
package body Pkg is
  procedure Absolute
    (X : in Integer;
     Y : out Integer)
  is begin
    if X < 0
      then Y := -X;
    else Y := X;
    end if;
  end Absolute;
end Pkg;

pkg.ads

pragma SPARK_Mode (On);
package Pkg is
  procedure Absolute
    (X : in Integer;
     Y : out Integer)
  with Pre => (X /= Integer'First),
        Post => (0 <= Y);
end Pkg;

shell

[willard@thor]$ gnatprove -P proj.gpr
Phase 1 of 3: frame condition computation ...
Phase 2 of 3: analysis and translation to intermediate language ...
Phase 3 of 3: generation and proof of VCs ...
analyzing Pkg, 0 checks
analyzing Pkg.Absolute, 1 checks
Summary logged in /home/willard/Documents/spark-lecture/abs0/abs11/gnatprove/gnatprove.out
[willard@thor]$
pragma SPARK_Mode (On);
package body Pkg is
  procedure Absolute
    (X : in Integer;
     Y : out Integer)
  is begin
    -- Y := |X|
    end Absolute;
  procedure Test
  is
    Z : Integer;
  begin
    Absolute(4, Z);
    pragma Assert (0 <= Z);
  end Test;
end Pkg;

pragma SPARK_Mode (On);
package Pkg is
  procedure Absolute
    (X : in Integer;
     Y : out Integer)
  with Pre  => (X /= Integer'First),
         Post => (0 <= Y);
end Pkg;
```
pragma SPARK_Mode (On);
package body Pkg is
  procedure Absolute
    (X : in Integer;
     Y : out Integer)
  is begin
    -- Y := |X|
    end Absolute;
  procedure Test
  is
    Z : Integer;
  begin
    Absolute(4, Z);
    pragma Assert (0 <= Z);
  end Test;
end Pkg;
```

```
pragma SPARK_Mode (On);
package Pkg is
  procedure Absolute
    (X : in Integer;
     Y : out Integer)
  with Pre  => (X /= Integer'First),
        Post => (0 <= Y);
end Pkg;
```

```
[willard@thor]$ gnatprove -P proj.gpr
Phase 1 of 3: frame condition computation ...
Phase 2 of 3: analysis and translation to intermediate language ...
pkg.ads:8:14: warning: subprogram "Test" has no effect
gprbuild: *** compilation phase failed
gnatprove: error during analysis and translation to intermediate language, aborting.
[willard@thor]$
```
pragma SPARK_Mode (On);
package body Pkg is
  procedure Absolute
    (X : in Integer;
     Y : out Integer)
  is begin
    -- Y := |X|
    end Absolute;
  Z : Integer;
  procedure Test
  is begin
    Absolute(4, Z);
    pragma Assert (0 <= Z);
  end Test;
end Pkg;

pragma SPARK_Mode (On);
package Pkg is
  procedure Absolute
    (X : in Integer;
     Y : out Integer)
  with Pre => (X /= Integer'First),
       Post => (0 <= Y);
end Pkg;
pkg.adb

```ada
pragma SPARK_Mode (On);
package body Pkg is
    procedure Absolute
        (X : in Integer;
         Y : out Integer)
    is begin
        -- Y := |X|
        end Absolute;
    Z : Integer;
    procedure Test
    is begin
        Absolute(4, Z);
        pragma Assert (0 <= Z);
    end Test;
end Pkg;
```

pkg.ads

```ada
pragma SPARK_Mode (On);
package Pkg is
    procedure Absolute
        (X : in Integer;
         Y : out Integer)
    with Pre  => (X /= Integer'First),
         Post  => (0 <= Y);
end Pkg;
```

shell

```
[willard@thor]$ gnatprove -P proj.gpr
Phase 1 of 3: frame condition computation ...
Phase 2 of 3: analysis and translation to intermediate language ...
Phase 3 of 3: generation and proof of VCs ...
analyzing Pkg, 0 checks
analyzing Pkg.Absolute, 2 checks
analyzing Pkg.Test, 2 checks
Summary logged in /home/willard/Documents/spark-lecture/abs0-bak/abs21/gnatprove/gnatprove.out
[willard@thor]$  
```
pragma SPARK_Mode (On);
package body Pkg is
    procedure Absolute
        (X : in Integer;
         Y : out Integer)
    is begin
        -- Y := |X|
        end Absolute;
    Z : Integer;
    procedure Test
        is begin
            Absolute(4, Z);
            pragma Assert (4 = Z);
        end Test;
end Pkg;

pragma SPARK_Mode (On);
package Pkg is
    procedure Absolute
        (X : in Integer;
         Y : out Integer)
    with Pre => (X /= Integer'First),
        Post => (0 <= Y);
end Pkg;
pkg.adb

pragma SPARK_Mode (On);
package body Pkg is
    procedure Absolute
        (X : in Integer;
         Y : out Integer)
    is begin
        -- Y := |X|
        end Absolute;
    Z : Integer;
    procedure Test
    is begin
        Absolute(4, Z);
        pragma Assert (4 = Z);
    end Test;
end Pkg;

pkg.ads

pragma SPARK_Mode (On);
package Pkg is
    procedure Absolute
        (X : in Integer;
         Y : out Integer)
    with Pre => (X /= Integer'First),
            Post => (0 <= Y);
end Pkg;

shell

$ gnatprove -P proj.gpr
Phase 1 of 3: frame condition computation ...
Phase 2 of 3: analysis and translation to intermediate language ...
Phase 3 of 3: generation and proof of VCs ...
analyzing Pkg, 0 checks
analyzing Pkg.Absolute, 2 checks
analyzing Pkg.Test, 2 checks
pkg.adb:16:7: warning: assertion might fail, requires 4 = Z
gprbuild: *** compilation phase failed
gnatprove: error during generation and proof of VCs, aborting.
$
pkg.adb

pragma SPARK_Mode (On);
package body Pkg is
  procedure Absolute
    (X : in Integer;
     Y : out Integer)
  is begin
    -- Y := |X|
  end Absolute;
  Z : Integer;
  procedure Test
  is begin
    Absolute(4, Z);
    pragma Assert (4 = Z);
  end Test;
end Pkg;

pkg.ads

pragma SPARK_Mode (On);
package Pkg is
  procedure Absolute
    (X : in Integer;
     Y : out Integer)
  with Pre => (X /= Integer'First),
            Post => (0 <= Y);
end Pkg;

why?

SPARK only knows pre/post conditions;
nothing more.
(and they don’t give us what we need)

shell

[willard@thor]$ gnatprove -P proj.gpr
Phase 1 of 3: frame condition computation ...
Phase 2 of 3: analysis and translation to intermediate language ...
Phase 3 of 3: generation and proof of VCs ...
analyzing Pkg, 0 checks
analyzing Pkg.Absolute, 2 checks
analyzing Pkg.Test, 2 checks
pkg.adb:16:7: warning: assertion might fail, requires 4 = Z
gprbuild: *** compilation phase failed
gnatprove: error during generation and proof of VCs, aborting.
[willard@thor]$
package body Pkg is
  procedure Absolute
    (X : in Integer;
     Y : out Integer)
  is begin
    -- Y := |X|
    end Absolute;
  Z : Integer;
  procedure Test
  is begin
    Absolute(4, Z);
    pragma Assert (4 = Z);
  end Test;
end Pkg;

package Pkg is
  procedure Absolute
    (X : in Integer;
     Y : out Integer)
  with Pre => (X /= Integer'First),
          Post => (0 <= Y and then
                    (if 0 <= X then X = Y));
  procedure Test;
end Pkg;
pragma SPARK_Mode (On);
package body Pkg is
  procedure Absolute
    (X : in Integer;
     Y : out Integer)
  is begin
    -- Y := |X|
    end Absolute;
  Z : Integer;
  procedure Test
  is begin
    Absolute(4, Z);
    pragma Assert (4 = Z);
  end Test;
end Pkg;

pragma SPARK_Mode (On);
package Pkg is
  procedure Absolute
    (X : in Integer;
     Y : out Integer)
  with Pre => (X /= Integer'First),
    Post => (0 <= Y and then
             (if 0 <= X then X = Y));
  procedure Test;
end Pkg;

shell
[willard@thor]$ gnatprove -P proj.gpr
Phase 1 of 3: frame condition computation ...
Phase 2 of 3: analysis and translation to intermediate language ...
Phase 3 of 3: generation and proof of VCs ...
analyzing Pkg, 0 checks
analyzing Pkg.Absolute, 2 checks
analyzing Pkg.Test, 2 checks
Summary logged in /home/willard/Documents/spark-lecture/abs0-bak/abs4/gnatprove/gnatprove.out
[willard@thor]$
package body Pkg is
   procedure Absolute
      (X : in Integer;
       Y : out Integer)
      is begin
      -- Y := |X|
      end Absolute;
      Z : Integer;
      procedure Test
      is begin
      Absolute(4, Z);
      pragma Assert (4 = Z);
      end Test;
   end Pkg;
pragma SPARK_Mode (On);

package body Pkg is
    procedure Absolute
        (X : in Integer;
         Y : out Integer)
    is begin
        -- Y := |X|
        end Absolute;
    Z : Integer;
    procedure Test
    is begin
        Absolute(4, Z);
        pragma Assert (4 = Z);
    end Test;
end Pkg;

package Pkg is
    procedure Absolute
        (X : in Integer;
         Y : out Integer)
    with Pre => (X /= Integer'First),
             Post => (0 <= Y
                           and then (if 0 <= X then X = Y)
                           and then (if X < 0 then Y = -X));
    procedure Test;
end Pkg;
pragma SPARK_Mode (On);
package body Pkg is
    procedure Absolute
        (X : in Integer;
         Y : out Integer)
    is begin
        -- Y := |X|
        end Absolute;
    Z : Integer;
    procedure Test
    is begin
        Absolute(4, Z);
        pragma Assert (4 = Z);
    end Test;
end Pkg;

pragma SPARK_Mode (On);
package Pkg is
    procedure Absolute
        (X : in Integer;
         Y : out Integer)
    with Pre => (X /= Integer'First),
             Post => (0 <= Y
                       and then (if 0 <= X then X = Y)
                       and then (if X < 0 then Y = -X));
    procedure Test;
end Pkg;

shell

[willard@thor]$ gnatprove -P proj.gpr
Phase 1 of 3: frame condition computation ...
Phase 2 of 3: analysis and translation to intermediate language ...
Phase 3 of 3: generation and proof of VCs ...
analyzing Pkg, 0 checks
analyzing Pkg.Absolute, 2 checks
analyzing Pkg.Test, 2 checks
Summary logged in /home/willard/Documents/spark-lecture/abs0-bak/abs4/gnatprove/gnatprove.out
[willard@thor]$
pragma SPARK_Mode (On);
package body Pkg is
  procedure Absolute
    (X : in Integer;
     Y : out Integer)
  is begin
    -- Y := |X|
  end Absolute;
  -- *expression* function (special)
  function AbsFun (X : Integer) return Integer is
    (if X < 0 then -X else X)
  with Pre => (X /= Integer'First);
  Z : Integer;
  procedure Test
  is begin
    Absolute(4, Z);
    pragma Assert (AbsFun(4) = Z);
  end Test;
end Pkg;
pragma SPARK_Mode (On);
package body Pkg is
  procedure Absolute
      (X : in Integer;
       Y : out Integer)
  is begin
     -- Y := |X|
     end Absolute;
  -- *expression* function (special)
  function AbsFun (X : Integer) return Integer is
     (if X < 0 then -X else X)
  with Pre => (X /= Integer'First);
  Z : Integer;
  procedure Test
  is begin
     Absolute(4, Z);
     pragma Assert (AbsFun(4) = Z);
  end Test;
end Pkg;

functions to define predicates; usable in pre/post conditions.

[willard@thor]$ gnatprove -P proj.gpr
Phase 1 of 3: frame condition computation ...
Phase 2 of 3: analysis and translation to intermediate language ...
Phase 3 of 3: generation and proof of VCs ...
analyzing Pkg.AbsFun, 1 checks
analyzing Pkg, 0 checks
analyzing Pkg.Absolute, 3 checks
analyzing Pkg.Test, 3 checks
Summary logged in /home/willard/Documents/spark-lecture/abs0-bak/abs6/gnatprove/gnatprove.out
[willard@thor]$
Summary: Software Analysis

- starting point: unsolvable problem.
- by being shrewd, we can still get correctness.
- proving correctness w/ contracts.
- example of a tool: SPARK.
  - easy to analyze; hard to write bad programs
  - fully-static enforcement, consisting of a
    - flow analysis
      - proves e.g. (in)dependence: noninterference
    - verification
      - proves much more, e.g. access control, ...