High Integrity Software Development (F29HD2)

Industrial Strength Program Proof

Andrew Ireland

Department of Computer Science
School of Mathematical and Computer Sciences
Heriot-Watt University
Edinburgh

SPARK and Program Proof

- As well as the **core annotations**, previously discussed, the SPARK Approach supports **proof annotations**:
  - pre- and postconditions,
  - auxiliary assertions, *e.g.* loop invariants,
  - declaration of proof functions, *i.e.* functions which play the role of specifications.
- The Examiner supports the automatic generation of verification conditions (VCs).
- SPARK proof tools:
  - Simplifier (automatic)
  - SPADE Proof Checker (interactive)

Switch Revisited

```
procedure Switch(X, Y: in out Integer)
--# derives X from Y &
--#         Y from X;
--# pre  true;
--# post X = Y~ and Y = X~;
is
   T: Integer;
begin
   T:=X;
   X:=Y;
   Y:=T;
end Switch;
```

Note that `Y~` denotes the initial value of `Y`, *i.e.* `Y~` is a ghost variable.

Switch VC

```
H1:   true
      ->
C1:   y = y .
C2:   x = x .
```

- Note that `H1` denotes a hypothesis. In general, there will be multiple hypotheses, *i.e.* `H1`, `H2`, ...
- The hypotheses are separated from the conclusion by the `->` symbol.
- A conclusion is split into conjuncts, where each conjunct is labelled, *i.e.* `C1`, `C2`, ...

Exception Freedom Proofs

- While the SPARK Approach supports partial correctness proofs, the greatest use of the tools is in generating **exception freedom proofs**.
- **Exception freedom proofs:** proving that when a program is executed no run-time exceptions will occur.
- To achieve this, the Examiner can be instructed to introduce assertions (proof annotations) corresponding to the places in the code where an Ada compiler inserts run-time checks, *e.g.* checking before a division that a divide-by-zero is not about to occur.
- By definition, SPARK eliminates many of the run-time exceptions that can be raised within Ada. However, **index**, **range**, **division** and **overflow** checks can still raise exceptions in SPARK code.

Integer Overflows

- An integer overflow occurs when an arithmetic operation attempts to generate a numeric value that exceeds the available storage space.
- Ariane 5 cost the European Space Agency 10 years and $7 billion to produce. In 1996, just 39 seconds into its maiden flight, Ariane 5 initiated its self-destruct mechanism – destroying itself and its payload of four expensive and un-insured scientific satellites.
- The shutdown (self-destruct) occurred when the guidance system's own computer tried to convert one piece of data – **the sideways velocity of the rocket** – from a 64-bit format to a 16-bit format. The number was too big, and an overflow error resulted. The conversion routine had worked fine for Ariane 4 but, unluckily, Ariane 5 was a faster rocket than Ariane 4 ...

Buffer Overflows

A buffer overflow occurs when a program stores data into a fixed length buffer that is too small for it.

> *"On 19 Oct 2000, hundreds of flights were grounded or delayed because of a software problem in the Los Angeles air-traffic control system. The cause was attributed to a controller in Mexico typing 9 (instead of 5) characters of flight-description data, resulting in a buffer overflow."*

The Risks Digest

Buffer Overflows: More Examples

A buffer overflow may cause a program to crash, but may be more seriously, they can be exploited by hackers.

> *"Security expert explains New York Times site break in ... They exploit these scripts to send a string of long commands to cause a buffer overflow that lets them into the operating system."*

CNN.com

> *"Buffer overflows have been the most common form of security vulnerability for the past 10 years."*

DARPA

A Simple Example

```
type T is range -128 .. 128;      procedure_inc_1.
procedure Inc(X: in out T)        H1:    true .
--# derives X from X;             H2:    x >= -128 .
is                                H3:    x <= 128 .
begin                                    ->
    X:= X+1;                      C1:    x + 1 >= -128
end Inc;                          C2:    x + 1 <= 128 .
```

Calling procedure `Inc` with a value of 128 will cause an overflow, *i.e.* raise a `Constraint_Error` exception. As a consequence, the associated exception freedom VC is unprovable, *i.e.* assuming that X is in the range $-128 \ldots 128$, then we can not prove $X + 1 \leq 128$.

Patch via Coding

```
type T is range -128 .. 128; procedure_inc_1.
procedure Inc(X: in out T)   H1:    true .
--# derives X from X;        H2:    x >= -128 .
is                           H3:    x <= 128 .
begin                        H4:    x <  128 .
   if X < T'Last then                ->
      X:= X+1;               C1:    x + 1 >= -128 .
   end if;                   C2:    x + 1 <= 128 .
end Inc;
```

Using **defensive programming** (above right) we can eliminate the potential overflow. However, in some sense the procedure no longer works, *i.e.* we can not prove `post X = X~+1`.

Patch via Precondition Strengthening

```
type T is range -128 .. 128;
procedure Inc(X: in out T)
--# derives X from X;
--# pre X < T'Last;
is
begin
   X:= X+1;
end Inc;
```

```
procedure_inc_1.
H1:    x < 128 .
H2:    x >= -128 .
H3:    x <= 128 .
        ->
C1:    x + 1 >= -128 .
C2:    x + 1 <= 128 .
```

We can add a precondition (above left), giving a provable VC, *i.e.* `X` is now strictly less that `128`. However, we now have to prove that every call to `Inc` satisfies the precondition.

Examiner & VCs

- Getting the Examiner to generate VCs:

  `spark -exp inc_value.ads inc_value.adb`

- VCs, and associated files, are written to a subdirectory of the current working directory.
- The subdirectory is named after the root name of the package being analyzed, and contains 3 files with extensions:

  vcg: contains the raw VCs.

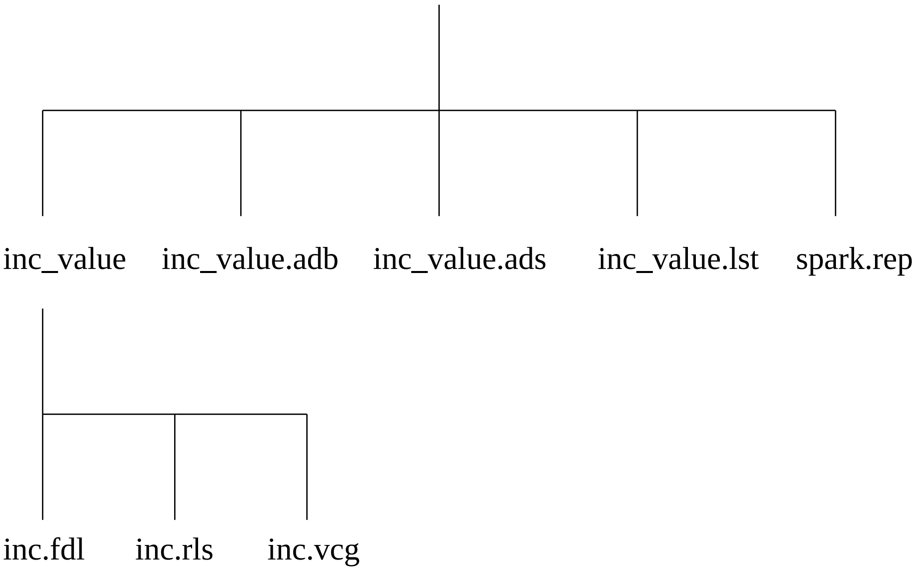
  fdl: contains information on the types of objects that appear within the VCs, *e.g.* functions and variables.

  rls: contains rules for manipulating the objects that appear within the VCs.

  Note: fdl stands for **functional description language**.

Examiner & VCs

Directory view:



Simplifier & VCs

- The Simplifier is applied within the subdirectory that contains the VC file, and is invoked using the command:

  `sparksimp`

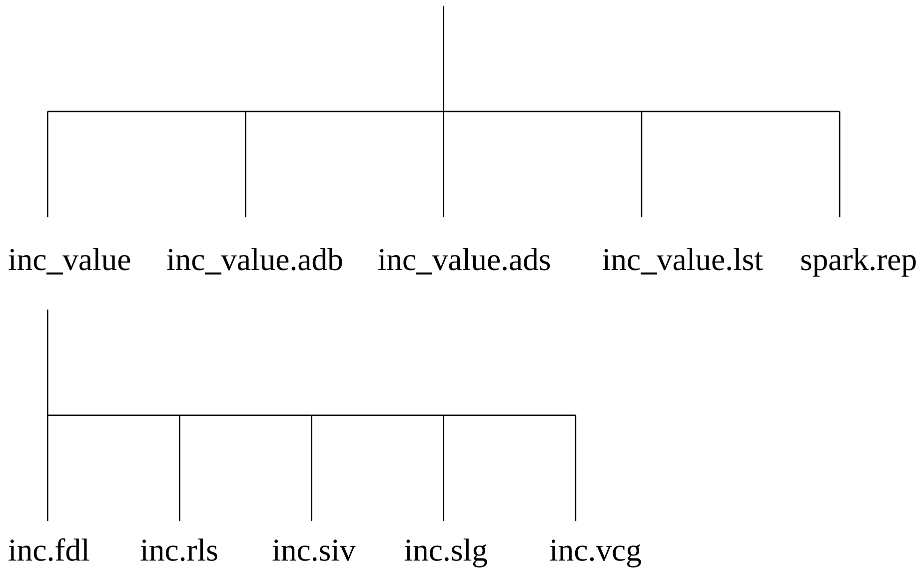
- The Simplifier generates 2 new files with extensions:

  siv: contains the simplified version of the raw VCs.

  slg: contains a log which summaries how the Simplifier discharged or simplified VCs.

Simplifier & VCs

Directory view:



Simplifier & VCs

```
procedure_inc_1.
H1:    x < t__last .
H2:    x >= t__first .
H3:    x <= t__last .
        ->
C1:    x + 1 >= t__first .
C2:    x + 1 <= t__last .
```

Raw VC

```
procedure_inc_1.
H1:    x < 128 .
H2:    x >= -128 .
H3:    x <= 128 .
        ->
C1:    x + 1 >= -128 .
C2:    x + 1 <= 128 .
```

Simplified VC

The above simplification step is performed by the Simplifier using the rules given with in the `rls` file, *i.e.* `inc.rls` includes the following rules:

```
...
inc_rules(4): t__first may_be_replaced_by -128.
inc_rules(5): t__last may_be_replaced_by 128.
...
```

Note that the Simplifier does not require these rules to discharge (prove) both `C1` and `C2`. In general, however, such rules are necessary.

Reviewing Proofs

- POGS – Proof ObliGation Summarizer
- There exist 4 ways in which VCs can be proved, *i.e.*
  - **Examiner:** will discharge the most trivial VCs, *i.e.* they are not presented in the **vcg** files.
  - **Simplifier:** will discharge most exception freedom VCs, typically success rate is around 94%.
  - **Proof Checker:** interactive proofs generate **plg** files.
  - **Review Team:** a proof review file, or **prv**, is the most informal form of proof evidence. It will typically be used where resources do not permit a formal proof from being constructed.

Reviewing Proofs

- POGS searches the current subdirectory for evidence of successful proofs, *e.g.* the `siv` file generated by the Simplifier, and generates a summary file with extension **sum**.
- POGS is invoked using the command:

  `pogs`

- The `pogs` summary generated for the `inc_value` example, `inc_value.sum`, is shown below:

```
VCs for procedure_inc :
   -------------------------------------------------------
     |     |                |  -Proved In-  |     |     |
   # |From | To             |vcg|siv|plg|prv|False|TO DO|
   -------------------------------------------------------
   1 |start|rtc check@7     |   |YES|   |   |     |     |
   2 |start|   assert@finish|YES|   |   |   |     |     |
   -------------------------------------------------------
```

```
Summary:

Total subprograms fully proved by examiner:                  0
Total subprograms fully proved by simplifier:                1
Total subprograms fully proved by checker:                   0
Total subprograms fully proved by review:                    0
Total subprograms with at least one false VC:                0
Total subprograms with at least one undischarged VC:         0
                                                         -----
Total subprograms for which VCs have been generated:         1

Total VCs by type:
                            -----------Proved By------------
               Total Examiner Simp Checker Review False Undscgd
Assert or Post:    1        1    0       0      0     0       0
Precondition check:0        0    0       0      0     0       0
Check statement:   0        0    0       0      0     0       0
Runtime check:     1        0    1       0      0     0       0
Refinement VCs:    0        0    0       0      0     0       0
Inheritance VCs:   0        0    0       0      0     0       0
===============================================================
Totals:            2        1    1       0      0     0       0
% Totals:                 50%  50%      0%     0%    0%      0%
```


Architectural Configurations

- Where a program makes use of basic types, *e.g.* `Integer`, the SPARK proof tools need to know how large an integer is, if exception freedom is to be guaranteed.
- This is achieved via a configuration, or **cfg**, file. A typical **cfg** file might look like:

```
package Standard is
   type Integer is range -2**31 .. 2**31-1;
   type Short_Short_Integer is range -2**7 .. 2**7-1;
   type Short_Integer is range -2**15 .. 2**15-1;
   type Long_Integer is range -2**31 .. 2**31-1;
   type Long_Long_Integer is range -2**63 .. 2**63-1;
end Standard;
```

The **cfg** file should reside in the directory that contains your source files.

Architectural Configurations

- In order to exploit a configuration file you need to reference it within the SPARK command line, *e.g.* if you are analyzing `foo.ads` and `foo.adb`, and the **cfg** file is called `gnat.cfg`, then you require the following command line:

```
spark -config_file=gnat.cfg -exp foo.ads foo.adb
```

- Assuming that `foo` makes use of the `Integer` type, then the generated **rls** file will include the following rules:

```
filterintege_rules(4):
        integer__first may_be_replaced_by -2147483648.
filterintege_rules(5):
        integer__last may_be_replaced_by 2147483647.
filterintege_rules(6):
        integer__base__first may_be_replaced_by -2147483648.
filterintege_rules(7):
        integer__base__last may_be_replaced_by 2147483647.
```


An Example

```
subtype AR_T is Integer range 0..9;
type A_T is array (AR_T) of Integer;
...
procedure Filter(A: in A_T; R: out Integer) is
begin
   R:=0;
   for I in AR_T loop
       if A(I)>=0 and A(I)<=100 then
           R:=R+A(I);
       end if;
   end loop;
end Filter;
```

Exception Freedom VC

```
H1: for_all (i___1: integer, ((i___1 >= ar_t__first) and
    (i___1 <= ar_t__last)) -> ((element(a, [i___1]) >=
      integer__first) and (element(a, [i___1]) <=
      integer__last))) .
H2: loop__1__i >= ar_t__first .
H3: loop__1__i <= ar_t__last .
H4: element(a, [loop__1__i]) >= 0 .
H5: element(a, [loop__1__i]) <= 100 .
H6: r >= integer__first .
H7: r <= integer__last .
    ->
C1: r + element(a, [loop__1__i]) >= integer__first .
C2: r + element(a, [loop__1__i]) <= integer__last .
C3: loop__1__i >= ar_t__first .
C4: loop__1__i <= ar_t__last .
```


A Closer Look

```
...
H4: element(a, [loop__1__i]) >= 0 .
H5: element(a, [loop__1__i]) <= 100 .
H6: r >= integer__first .
H7: r <= integer__last .
   ->
...
C2:r+element(a,[loop__1__i])<=integer__last
...
```

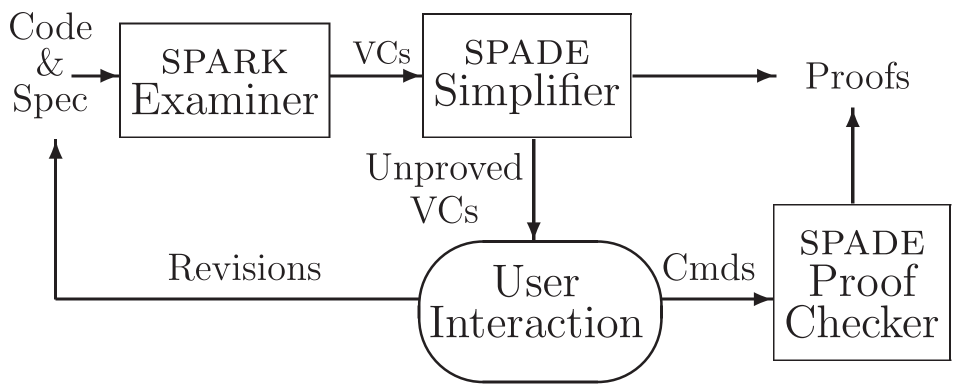
Note: `element(X, [Y])` denotes the Y^{th} element of array X.

A Closer Look

```
...
H4: element(a, [loop__1__i]) >= 0 .
H5: element(a, [loop__1__i]) <= 100 .
H6: r >= integer__first ← −32768 .
H7: r <= integer__last ←— 32767 .
   ->
...
C2:r+element(a,[loop__1__i])<=integer__last
...                         ↑
```

VC is unprovable, *i.e.*
if (32668 $\leq$ *r* $\leq$ 32767) then possible overflow error

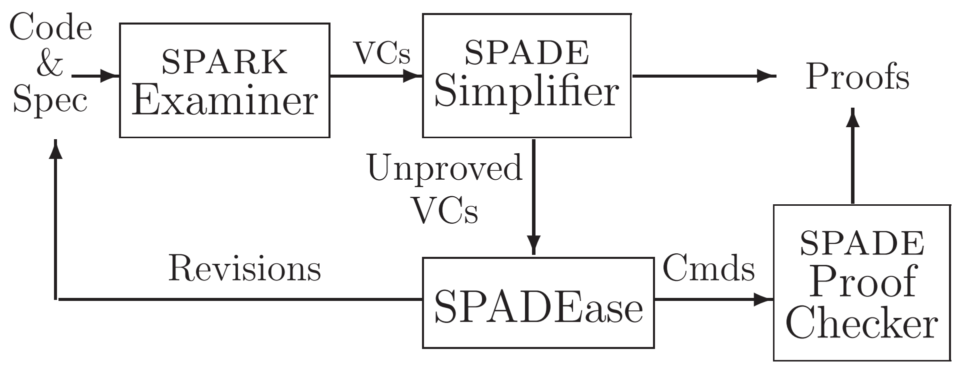
The SPARK Approach



Patch: Loop Invariant Strengthening

```
procedure Filter(A: in A_T; R: out Integer) is
begin
   R:=0;
   for I in AR_T loop
   --# assert R >= 0 and R <= I*100;
      if A(I)>=0 and A(I)<=100 then
          R:=R+A(I);
      end if;
   end loop;
end Filter;
```

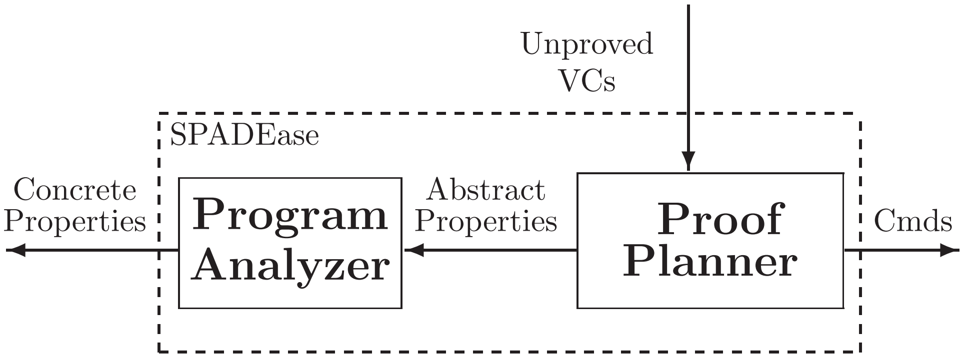

Extending the SPARK Approach



SPADEase

- SPADEase = Proof Planning + Program Analysis
- Proof Planning uses high-level proof outlines, known as proof plans, to guide proof automation.
- Program Analysis is used to generate relatively simple program properties directly from the source code, *i.e.* constraints on variables.
- SPADEase promotes a cooperative approach to program reasoning, *i.e.* partial success in the search for a proof is used to selectively introduce program properties via proof annotations.

SPADEase



Revised Exception Freedom VC

```
H1: r >= 0 .
H2: r <= loop__1__i * 100 .
...
H6: element(a, [loop__1__i]) >= 0 .
H7: element(a, [loop__1__i]) <= 100 .
...
   ->
...
C2:r+element(a,[loop__1__i])<=integer__last
...
```


Proof Planning: Exception Freedom

Given: $r \leq i * 100 \ \wedge \ ele(a,i) \leq 100$

Proof: $r + ele(a,i) \leq integer\_last$

`transitivity`

$$r + ele(a,i) \leq X_0 \wedge X_0 \leq integer\_last$$

`decomposition`

$$r \leq X_1 \wedge ele(a,i) \leq X_2 \wedge X_1 + X_2 \leq integer\_last$$

`fertilize`

$$((i * 100) + 100) \leq integer\_last$$

`simplify`

$$((i * 100) + 100) \leq 32767$$

`elementary`

Note: `fertilize` produces $\{i * 100/X_1, \ 100/X_2\}$

Loop Invariant VC

```
H1: r >= 0 .
H2: r <= loop__1__i * 100 .
...
H6: element(a, [loop__1__i]) >= 0 .
H7: element(a, [loop__1__i]) <= 100 .
...
   ->
C1:r+element(a,[loop__1__i]) >= 0 .
C2:r+element(a,[loop__1__i]) <=
                   (loop__1__i + 1) * 100 .
...
```


Proof Planning: Loop Invariant

Given: $r \leq i * 100 \ \wedge \ ele(a,i) \leq 100$

Proof: $r + ele(a,i) \leq (i+1) * 100$

`annotate`

$$\boxed{\boxed{r} + ele(a,i)}^{\uparrow} \leq \boxed{(\boxed{i} + 1)}^{\uparrow} * 100$$

`wave`

$$\boxed{\boxed{r} + ele(a,i)}^{\uparrow} \leq \boxed{(\boxed{i * 100}) + (1 * 100)}^{\uparrow}$$

`wave`

$$\boxed{\boxed{r \leq i * 100} \wedge ele(a,i) \leq 1 * 100}^{\uparrow}$$

`fertilize`

$$true \wedge ele(a,i) \leq 1 * 100$$

`simplify`

$$ele(a,i) \leq 100$$

`elementary`

SPADEase Results

- Our evaluation was based upon examples drawn from the literature and industrial data provided by Praxis, *e.g.* SHOLIS
- SPADE Simplifier is very effective on exception freedom VCs, *i.e.* typical hit-rate of 90%
- SPADEase targeted the VCs which the SPADE Simplifier failed to prove *i.e.* typically loop-based code
- While critical software is engineered to minimize the number and complexity of loops, we found that 80% of the loops we encountered were provable using our techniques

Summary

Learning outcomes:

- Understand how program proof fits within the SPARK Approach.
- Understand exception freedom proof, and why it is of importance to industry.
- Be able to interpret simple VCs generated by the Examiner.
- Be able to use the Simplifier to discharge VCs.
- Be able to use POGS to interpret the results of the Simplifier.

Summary

Recommended reading:

- R. Chapman and P. Amey, "Industrial Strength Exception Freedom", Proceedings of ACM SigAda, 2002.
- A. Ireland and B.J. Ellis and A. Cook and R. Chapman and J. Barnes, "An Integrated Approach to High Integrity Software Verification" Journal of Automated Reasoning: Special Issue on Empirically Successful Automated Reasoning, Kluwer, Vol 36(4), 2006. (see also MACS Technical Report HW-MACS-TR-0027: `http://www.macs.hw.ac.uk:8080/techreps/`)
- SPADEase related web pages:
  `http//www.macs.hw.ac.uk/nuspade`
  `http//www.macs.hw.ac.uk/spadease`