NuSPADE: An Integrated Approach to Exception Freedom Proof

Andrew Ireland

Dependable Systems Group
School of Mathematical and Computer Sciences
Heriot-Watt University
Edinburgh

Context

- Investigate the role of proof planning within the SPARK Approach to high integrity software
- Bill Ellis (Research Associate)
- Funded by the EPSRC Critical Systems Programme (GR/R24081) in collaboration with Praxis

`www.macs.hw.ac.uk/nuspade`

Overview

- Focus on the problems that arise when proving exception freedom verification conditions (VCs)
- Present an integrated approach to these problems
- Results and future directions

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 .
```

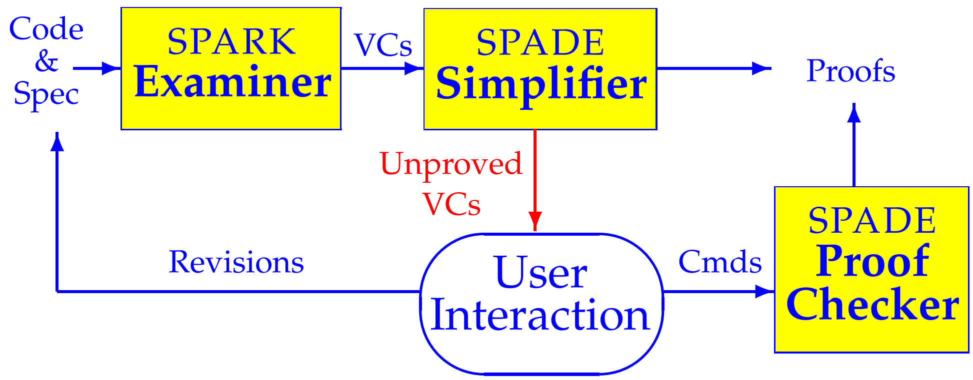

A Closer Look

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

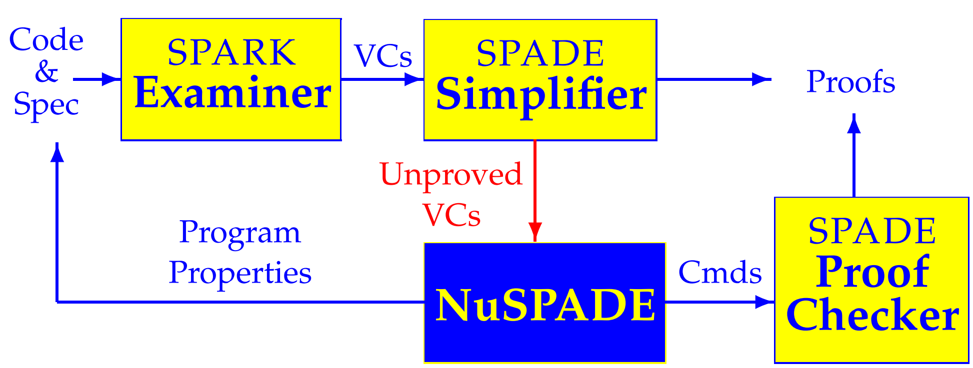
VC is unprovable, *i.e.*

if (32668 $\leq r \leq$ 32767) then possible overflow error

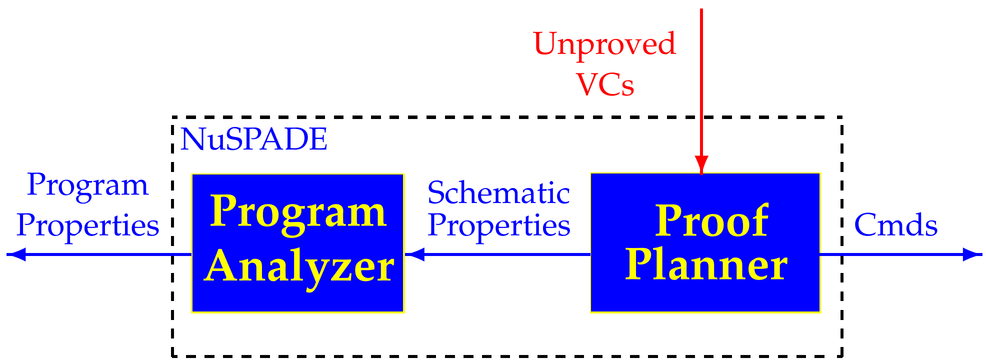
The SPARK Approach



Extending the SPARK Approach



NuSPADE



Proof Planning

- Use of high-level proof outlines, known as proof plans, to guide proof search
- proof plan = methods + critics + tactics
- Proof planning supports:
  - a flexible style of proof search, *e.g.* use of meta-variables to delay choice during proof search
  - automatic proof patching via proof-failure analysis
  - diversity of proof, *i.e.* tactics can be ported to different proof checkers

Proof Methods

Exception Freedom	Loop Invariant
`elementary`	`annotate`
`simplify`	`wave`
`transitivity`	`induction`
`decomposition`	`generalize`
`fertilize`	

Program Analysis

- Focus on the generation of program properties that support exception freedom proof
- Program analysis methods:
  - `subtype`
  - `non-looping code`
  - `looping code`
  - `loop guards`

Integration via Proof-Failure Analysis

- A proof method is applicable if all its preconditions are true
- A proof critic is applicable if its associated proof method fails to apply
- Proof-failure analysis is used to guide the selection of program properties that will progress the proof process

Proof-Failure Analysis

Preconditions for elementary **critic:**

- All preconditions for the elementary method fail.
- There exists a top-level goal of the form $E\ Rel\ C$, *e.g.*

  $r + element(a, i) \leq integer\_last$
- There exists a variable V_i that occurs within E such that there exists a hypothesis of the form $V_i\ Rel\ E_i$, *e.g.*

  $r \leq integer\_last$
- A counter-example can be found that shows that the bound E_i is insufficient to prove exception freedom, *e.g.*

  $32668 \leq r \leq 32767$

Patch: generate schematic properties, *e.g.* $(r \geq X) \wedge (r \leq Y)$

A Program Analysis Technique

- Recurrence relations are recursive definitions of mathematical functions or sequences, *e.g.*

$$\begin{cases} g(0) &= 0 \\ g(n) &= g(n-1) + (2 * n) - 1 \end{cases}$$

- Solving a recurrence relation corresponds to finding a "closed form" of the function, *e.g.* $g(n) = n^2$
- Recurrence relations can be used to express the value of variables within loops, where the solutions provide loop invariants
- There are many off-the-shelf recurrence relation solvers, *e.g.* PURRS (University of Parma)

Recurrence Relation for Variable `I`

Recurrence Relation

$$\begin{cases} I_0 &= 0 \\ I_n &= I_{n-1} + 1 \end{cases}$$

Solution

$$I_n = \underline{I_0 + n}$$

$$I_0 \Rightarrow 0$$

$$I_n = \underline{0 + n}$$

$$0 + Y \Rightarrow Y$$

$$I_n = n$$

Recurrence Relation for Variable R

Recurrence Relation

$$\begin{cases} R_0 = 0 \\ R_n = R_{n-1} \\ R_n = R_{n-1} + \underbrace{ele(A, I)}_{\text{problem term}} \end{cases}$$

Extreme Recurrence Relation

$$\begin{cases} R_0 = 0 \\ R_n = R_{n-1} \\ R_n = R_{n-1} + 0 \\ R_n = R_{n-1} + 100 \end{cases}$$

Note problem term is eliminated by generalizing the recurrence relation, *i.e.* by considering the bounds of $ele(A, I)$.

Solving for Variable R

- **true**-branch:

lower − bound	**upper − bound**
$R_n = R_{n-1} + 0$	$R_n = R_{n-1} + 100$
$R_n = R_0 + n * 0$	$R_n = R_0 + n * 100$
$R_n = 0$	$R_n = n * 100$

- **false**-branch:

$$R_n = R_{n-1}$$
$$R_n = R_0$$
$$R_n = 0$$

Combining Solutions

$$\underline{I_n = n \ \wedge \ (R_n = 0 \ \vee \ ((R_n \geq 0) \ \wedge \ (R_n \leq n * 100)))}$$

$$\Downarrow$$

$$\underline{I_n = n \ \wedge \ (R_n \geq 0) \ \wedge \ (R_n \leq n * 100)}$$

$$\Downarrow$$

$$(R_n \geq 0) \ \wedge \ (R_n \leq I_n * 100)$$

$$\Downarrow$$

```
--# assert R >= 0 and R <= I*100;
```


Revised Filter Code

```
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;
```

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
...
```


Planning Proof

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 .
...
```


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%
- NuSPADE 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

Phase	**Goal**	**Plan**		**Critic**				**PropGen**		
		a1	**a2**	**b1**	**b2**	**b3**	**b4**	**c1**	**c2**	**c3**
P1	P1_1	○		●				●		
	P1_2	○		●				●		
	P1_3		○		●				●	
	P1_4		○				●			●
	P1_5		○				●			●
P2	P2_1	●								
	P2_2	●								
	P2_3		●							
	P2_4		●							

Note: the key to the results table appears at the end of these notes

Limitations & Future Work

- Constraint solving fails when reasoning with "big numbers", *i.e.* integers out with $-(2^{25}) \ldots 2^{25} - 1$
- Precondition strengthening would improve our hit-rate, constraint solving may have a role to play
- We could make greater use of constraint solving for debugging
- Integrating decision procedures within NuSPADE would also improve our hit-rate
- A follow-on "knowledge transfer" project, funded by the EPSRC RAIS Scheme starts early 2005
- "Critical Software Components in SPARK", in collaboration with Kung-Kiu Lau (University of Manchester) and Praxis

Conclusion

- NuSPADE = Proof Planning + Program Analysis
- Proof planning guides proof search
- Proof-failure analysis coupled with program analysis selectively strengthens program specifications
- NuSPADE increases automation for exception freedom proof

Results Table Key

Plan	**a1**	**loop invariant**
	a2	**exception freedom**
Critic	**b1**	**fertilize**
	b2	**elementary**
	b3	**transitivity**
	b4	**decomposition**
PropGen	**c1**	**entry**
	c2	**for-loop range**
	c3	**range constraint**

Note that • denotes the successful application of a proof plan while ◦ denotes partial success