Workshop on Trusted Extensions of Interactive Theorem Provers

Degrees of trustworthiness: observations arising from the SPARK proof tools and their use

Ian O'Neill

Topics

- Background: SPARK and its proof tools
- Examples of real verification conditions
- Extensions to the power of the proof tools
- Further extension: user-defined proof rules
- Current limitations: soundness and floating-point

SPARK

- A high-integrity subset of Ada
- Developed late 1980s/early 1990s onwards
- Contracts enforced by annotations
  - ‘Formal comments’
  - Ignored by a compiler, used by SPARK tools
- <u>Example:</u>

```
--# derives Temperature from Pressure, Volume;
--# pre Pressure in Valid_Pressure_T and
--#     Volume > 0.0;
```

SPARK's Proof Tools

- SPARK Proof Checker (interactive, short-rein)
  - Developed in Prolog *(formerly SPADE Proof Checker)*
  - Some 'oracles', e.g.
    - standardise a+2*(b-a)+3 = 1-a+2*(b+1) yields true.
    - element(update(a,[3],x),[2]) simplifies to element(a, [2]).
- First industrial use of Checker to prove LUCOL assembly code modules for RB211-524G met their specification (1986/87)
- Simplifier (non-interactive, 'batch' operation)
- Simplifier 'derived' from Checker components

Soundness

- Soundness of original Checker:
  - Components: standardiser, expression simplifier, rules engine, natural deduction strategies
  - 'Boot-strapping' process:
    - Establish soundness of standardiser by induction
    - Use in proving soundness of expression simplifier
    - Then other components, and so on
  - Proofs only to establish soundness, not completeness or termination
- Soundness of original Simplifier:
  - Stringing together of sound Checker components

Topics

- Background: SPARK and its proof tools
- Examples of real verification conditions
- Extensions to the power of the proof tools
- Further extension: user-defined proof rules
- Current limitations: soundness and floating-point

Verification Conditions

- Advent of SPARK and Examiner:
  - VCs generated for multiple units
  - Proofs of exception-freedom
    - Exception-freedom VCs tend to be simpler, but
    - Much more numerous
- Led to decision to create standalone Simplifier:
  - Most exception-freedom VCs discharge automatically
  - Remainder: can be discharged with Checker, with another trusted proof tool or by hand
  - But: with proof by hand, risk of misproof

Real Example VC: range constraint

```
H1:   ...
H68:  fld_value(s__cr) >= basictypes__rollt__first .
H69:  fld_value(s__cr) <= basictypes__rollt__last .
...
H72:  ...
      ->
C1:   abs(fld_value(s__cr)) >= basictypes__rollt__base__first .
C2:   abs(fld_value(s__cr)) <= basictypes__rollt__base__last .
```

Larger, more complex subprograms yield more hypotheses, more VCs to show each subexpression is within relevant range, etc.

Example VC: structured types

- A (relatively) simple correctness VC from SPARK test set for an array of records:

```
H1:   true .
H2:   for_all (i___1: natbyte, ((i___1 >= it1__first) and (i___1 <= it1__last)) -> ((fld_g1(element(a, [
      i___1])) >= et2__first) and (fld_g1(element(a, [i___1])) <= et2__last))) .
H3:   for_all (i___1: natbyte, ((i___1 >= it1__first) and (i___1 <= it1__last)) -> ((fld_f1(element(a, [
      i___1])) >= et1__first) and (fld_f1(element(a, [i___1])) <= et1__last))) .
H4:   i >= it1__first .
H5:   i <= it1__last .
H6:   f >= et1__first .
H7:   f <= et1__last .
H8:   f >= et1__first .
H9:   f <= et1__last .
H10:  i >= it1__first .
H11:  i <= it1__last .
    ->
C1:   for_all (n_: natbyte, ((n_ >= it1__first) and (n_ <= it1__last)) -> (true and (((fld_f1(element(update(
      a, [i], upf_f1(element(a, [i]), f)), [n_])) >= et1__first) and (fld_f1(element(update(a, [i], upf_f1(element(
      a, [i]), f)), [n_])) <= et1__last)) and ((fld_g1(element(update(a, [i], upf_f1(element(a, [i]), f)), [n_])) >=
      et2__first) and (fld_g1(element(update(a, [i], upf_f1(element(a, [i]), f)), [n_])) <= et2__last))))) .
```

Example VC from Tokeneer

```
H12:   for_all(i___1 : integer, 1 <= i___1 and i___1 <= 17 ->
          0 <= element(logfileentries, [i___1]) and
          element(logfileentries, [i___1]) <= 1024) .
H13:   currentlogfile >= 1 .
H14:   currentlogfile <= 17 .
H16:   fld_length(usedlogfiles) <= 17 .
H22:   element(logfileentries, [currentlogfile]) <> 1024 or
          fld_length(usedlogfiles) <> 17 .
    ->
C1:    element(logfileentries, [currentlogfile]) < 1024 or
          fld_length(usedlogfiles) < 17 .
```

Reasoning too tortuous for Simplifier

Topics

- Background: SPARK and its proof tools
- Examples of real verification conditions
- Extensions to the power of the proof tools
- Further extension: user-defined proof rules
- Current limitations: soundness and floating-point

Trusted extensions to tools

- VCs from real projects (SHOLIS and Tokeneer) which weren't discharged automatically, but which were provable, were reviewed for common patterns.
- These were used to identify potential improvements:
  - Arithmetic reasoning (abs, division, modulus, exponentiation, special cases)
  - Logic automation (e.g. better tactics for implication and disjunction conclusions)
  - Improved handling of structured objects

Trustworthiness of extensions (1)

- Identify new inference rules which will improve Simplifier ‘hit rate’
  - generalising, based on examples identified
  - determine expected impact of changes (this is approximate, based on nature of improvements and ‘gut feel’ from categorising each VC)
- Prove that these rules are sound with the Checker
  - manual process to generate VCs
  - review to check the VCs correspond to the rules
  - formal proof of the VCs with the Checker
- Add these proofs to the standard SPARK test set

Trustworthiness of extensions (2)

- Incorporate the new rules into the Simplifier
- Add extra tests which are unprovable
  - E.g. variants of provable VCs with each of the necessary hypotheses omitted in turn
- Run through entire test set
- Confirm expected results achieved
  - Investigate mismatches:
    - VCs unexpectedly not proved
    - VCs unexpectedly proved
    - Any other changes (e.g. partial proofs)
  - Update test set in light of improved results

Example of improvement

```
% Div(22): X - X div Y * Y <= N may_be_deduced_from [(1) X >= 0,
%                                                    (2) Y > 0,
%                                                    (3) {X <= N | Y - 1 <= N} one of].
try_new_deduction_strategies(X - XdivYtimesY <= N, integer, Hs) :-
   i_am_using_rule(div_22a),
   (
      XdivYtimesY = X div Y * Y        ;   XdivYtimesY = Y * (X div Y)
   ),
   safe_deduce(X >= 0, integer, H1),        /* (1) */
   (                                        /* (2) */
      safe_deduce(Y > 0, integer, H2)    ;   safe_deduce(Y >= 1, integer, H2)
   ),
   (                                        /* (3) */
      safe_deduce(X <= N, integer, H3)   ;   safe_deduce(Y - 1 <= N, integer, H3)
   ),
   append(H2, H3, Hrest),
   append(H1, Hrest, HL),
   sort(HL, Hs).
```

VC proved to establish soundness

```
% Div(22): X - X div Y * Y <= N may_be_deduced_from
%                                   [(1) X >= 0,
%                                    (2) Y > 0,
%                                    (3) {X <= N | Y - 1 <= N} one of].
```

```
H1:   x >= 0.
H2:   y > 0
H3:   x <= n or y - 1 <= n.
->
  C1:   x - x div y * y <= n.
```

Can be proved by cases with the Checker

Extensions: results achieved

- Arithmetic reasoning improvements:
  - 235 additional SHOLIS/Tokeneer VCs were expected to be proved automatically
  - 248 were actually proved
  - other minor improvements; all were reviewed
- Structured objects improvements:
  - 188 additional VCs were expected to be proved when changes planned, but not all changes were made
  - 195 were actually proved; again any other improvements or deviations were also reviewed

Topics

- Background: SPARK and its proof tools
- Examples of real verification conditions
- Extensions to the power of the proof tools
- Further extension: user-defined proof rules
- Current limitations: soundness and floating-point

Extension to add user-defined rules

- Allow users to define additional inference and rewrite rules
  - Advantages: Simplifier user-extendable; user can write rules which capture reasoning and can be replayed/reused
  - Disadvantages: user can write unsound rules; potential new problems, e.g. termination
- User can tackle risk of unsoundness by process (formal proof of soundness of new rules), but tools do not enforce this
- Can tackle other issues internally: e.g. depth limit to prevent non-termination, etc.

User-defined proof rules: pragmatics

- Used as a 'last resort':
  - Simplification proceeds in a number of phases
  - User-defined rule application is tried last, only if a VC has not been fully discharged by other means
  - Use of rule(s) is documented in tool output
- Strict constraints on application:
  - Pattern matching
  - Discharge of ground / non-ground side-conditions
  - Driven primarily by structure of goal formula(e)

Real example: user-defined rule

- Unsound example (found by review):

  X <> 0 may_be_deduced_from

  [abs(X) >= Z, Z <> 0].

  - Written to discharge a specific VC
  - Not sound: let X = 0, Z = -1
  - Resolve by strengthening side-condition to Z > 0
  - Alternative to finding by review: try to construct proof with Proof Checker of formula

    (abs(x) >= z and z <> 0) -> x <> 0
  - (Can't be done: user spots defect.)

Topics

- Background: SPARK and its proof tools
- Examples of real verification conditions
- Extensions to the power of the proof tools
- Further extension: user-defined proof rules
- Current limitations: soundness and floating-point

Current limitations

- Proofs are only as sound as the user-defined proof rules that they use
- Floating-point numbers and proof:
  - We do not explicitly model Ada's real types
  - We use an abstraction: the mathematical reals
  - SPARK floating-point literals are represented as rational literals in VCs, e.g. 3.5 is modelled as 7/2.
  - It is possible to prove the code fragment

    ```
    X := 1.0 / 3.0;
    ```

    satisfies the postcondition

    ```
    --# post 3.0 * X = 1.0;
    ```

Floating-point limitations example

- Altitudes are input and displayed in (integer) feet
- Calculations use (floating-point) metres

```
Firm_Lower_Bound : constant := 0.0;     --  metres
Firm_Upper_Bound : constant := Altitudes.Max_Altitude_T *
     Units.Foot_As_Metres;                          --  metres

type Metres_T is digits 6 range
     Firm_Lower_Bound .. Firm_Upper_Bound;
```

Floating-point limitations example

- **Problem:** maximum input altitude is 67,000 feet, giving Firm_Upper_Bound of 20,421.6 metres.
- This is not a model number.
- Conversion from feet to metres can yield a constraint error at the boundary.
- **Solution:** add a small, type-dependent Epsilon:

```
type Metres_T is digits 6 range
    Firm_Lower_Bound ..
    (Firm_Upper_Bound + Epsilon.Digits_6_Range_1_E_4);
```

Floating-point limitations example

- **New problem:** with a non-zero Epsilon, we cannot prove VCs involving the conversions from the larger range to the smaller, typically. But if Epsilon is zero, we can't guarantee a constraint error won't be raised.
- **Solution:**
  - 'Pretend' Epsilon is zero for proof purposes (this can be done by using a SPARK 'shadow' package)
  - Use proper, non-zero value for compilation
  - Use Ada pragma to demonstrate there is no problem at compile-time…

Floating-point limitations example

```
pragma Compile_Time_Error (
        Metres_T'Model (Metres_T'First) > Firm_Lower_Bound or
        Metres_T'Model (Metres_T'Last)  < Firm_Upper_Bound,
        "Constraint_Error could be raised for this type.");
```

- Now, type Metres_T is slightly larger, including the model number after 20,421.6 metres, so a calculation that yields a value equivalent to exactly 20,421.6m (67,000ft) will not raise an exception. The Epsilon is chosen based on the type range, and will not accommodate a value equivalent to 67,001ft.

Conclusion

- Original work on establishing soundness of Checker still intact
- Reused components to generate Simplifier
- Extensions introduced in a controlled way, with proofs of soundness of new rules, peer review, additional testing and regression testing
- User-defined proof rules: a mixed blessing, in that unsound rules may in principle be used; need to put process in place to avoid this
- Limitations, e.g. in floating-point reasoning, can sometimes be addressed outside formal proof

Document Control

Change History

0.1 30/07/2010 First draft for comments

Originator: Ian O'Neill

Approver : Rod Chapman

Altran Praxis Limited

20 Manvers Street
Bath BA1 1PX
United Kingdom
Telephone: +44 (0) 1225 466991
Facsimile: +44 (0) 1225 469006
Website: www.altran-praxis.com

Email: ian.o'neill@altran-praxis.com