Deductive Verification in Frama-C and SPARK2014: Past, Present and Future

Claude Marché (Inria & Université Paris-Saclay)

OSIS, Frama-C & SPARK day, May 30th, 2017
Outline

Why this joint Frama-C and SPARK day?
common history of Frama-C and SPARK

ACSL and SPARK 2014: how they differ?
static versus runtime checking
specification languages: design choices
advertisement: ghost code

Recent and Future Trends
bit-wise, floating-point computations
proof debugging, counterexamples
interactively discharging VCs
Outline

Why this joint Frama-C and SPARK day?
common history of Frama-C and SPARK

ACSL and SPARK 2014: how they differ?
static versus runtime checking
specification languages: design choices
advertisement: ghost code

Recent and Future Trends
bit-wise, floating-point computations
proof debugging, counterexamples
interactively discharging VCs
Around 1990

Deductive Verification

- Formal Specification of functional behaviors using *contracts*
- Generation of *Verification Conditions*
- Computer-Assisted *Theorem Proving*

- **SPARK Examiner** for Ada’83
  - Univ. Southampton, Praxis, then Altran
  - home-made VC generator, simplifier, checker

- **CAVEAT**, static analyzer for C code
  - CEA
  - home-made VC generator and solver
Around 2000

- The *Why* tool for deductive verification
  - team ProVal (Inria & CNRS & Univ. Paris Sud)
  - a ML-style programming language with contracts
  - VC discharged using *Coq*
    - then later with *Simplify*
    - then with *Alt-Ergo*
    - then with several others

- Why front-ends:
  - for Java: *Krakatoa*
    - annotations \(\simeq\) *Java Modeling Language*
  - for C: *Caduceus*
    - annotation language \(\simeq\) JML
2005-2010

- **Frama-C**
  - CEA and ProVal
  - **ACSL** language
  - *plug-in architecture* to support various kind of analyses
- **Jessie**
  - Deductive Verification plug-in
  - Use Why as intermediate language
  - **Alias analysis** using memory regions
2010-2014

- **Why3**, new generation of Why
  - module system, rich standard library of theories
  - region-based type system for *alias control*
  - generic architecture to plug in back-end provers
- Jessie plug-in adapted to Why3
- **WP** Frama-C plug-in
  - various *memory models* and *aliasing conditions*
  - call provers through Why3
- **SPARK 2014**: SPARK new generation
  - AdaCore - Altran
  - Why3 as intermediate programming language
  - *Non-aliasing conditions* to ease VC generation and proof
  - call provers through Why3
Why3 ‘ecosystem’ today

- **C programs**
  - Frama-C
- **Ada programs**
  - SPARK 2014
- **Why3**
- **SMT solvers**
  - Alt-Ergo
  - CVC4
  - veriT
  - Z3
  - etc.
- **Interactive provers**
  - Coq
  - Isabelle
  - PVS
- **Other provers**
  - E prover
  - Gappa
  - SPASS
  - Vampire
  - etc.
Outline

Why this joint Frama-C and SPARK day?
common history of Frama-C and SPARK

ACSL and SPARK 2014: how they differ?
static versus runtime checking
specification languages: design choices
advertisement: ghost code

Recent and Future Trends
bit-wise, floating-point computations
proof debugging, counterexamples
interactively discharging VCs
Static versus Runtime Checking

Contracts can be used either

- for *runtime assertion checking* (RAC):
  - assertions executed and checked valid during execution, tests
- for *static verification* (VC generation + theorem proving)
  - code can be proved correct w.r.t. contracts

Example: Java Modeling Language

- JML RAC: turns assertions into regular Java code
- Static verification: *ESC/Java*, using solver *Simplify*
From JML to Krakatoa and ACSL

- JML was designed with RAC in mind
- Consequence: assertions are *Java Boolean expressions*
- Extensions to Java expressions: meant to be *executable*
  - e.g. quantification must be bounded
    
    \[
    (\forall \text{int } i; \ 0 \leq i \ \&\& \ i < a.\text{length}; \ P(i))
    \]

- Models for specifications can be designed using extra *pure* classes
  - methods need to be *terminating*
  - they *should not raise exceptions*
  - they *should not have side-effects*
Specification language is classical first-order logic with
- types (polymorphism)
- equality, built-in arithmetic
- user-defined theories to design abstract models
  - introducing new data-types, logic functions, predicates
  - either defined or axiomatized

Specification language

- distinct from programming language
- adequate for use of external provers
- does not need to be executable
Example: sorting algorithms

/*@ requires \valid(a+(0..n-1));
@ assigns a[0..n-1];
@ ensures sorted(a,0,n);
@ ensures permut{Pre,Post}(a,0,n-1);
@*/
void sort(int *a, int n) {

Example: sorting algorithms

/*@ predicate sorted(int *a, integer l, integer h) =
   @ \forall integer i j; l <= i <= j < h ==> a[i] <= a[j] ;
   @*/

- Not executable a priori
- Could be executed if ranges of i and j are somehow 'computed'
  - In JML, it should be written
    
    (\forall int i; l <= i && i < h ;
     (\forall int j; i <= j && j < h; a[i] <= a[j])))

- Notice also the type integer for mathematical, unbounded integers
Example: sorting algorithms

/*@ predicate swap{L1,L2}(int *a, integer i, integer j) =
  @ \at(a[i],L1) == \at(a[j],L2) && \at(a[j],L1) == \at(a[i],L2) &&
  @ \forall integer k; k != i && k != j ==> \at(a[k],L1) == \at(a[k],L2);
  @
  @ inductive permut{L1,L2}(int *a, integer l, integer h) {
    @ case permut_refl{L}:
    @  \forall int *a, integer l h; permut{L,L}(a, l, h);;
    @ case permut_sym{L1,L2}:
    @  \forall int *a, integer l h;
    @   permut{L1,L2}(a, l, h) ==> permut{L2,L1}(a, l, h);
    @ case permut_trans{L1,L2,L3}:
    @  \forall int *a, integer l h;
    @   permut{L1,L2}(a, l, h) && permut{L2,L3}(a, l, h) ==> permut{L1,L3}(a, l, h);;
    @ case permut_swap{L1,L2}:
    @  \forall int *a, integer l h i j;
    @   l <= i < h && l <= j < h && swap{L1,L2}(a, i, j) ==> permut{L1,L2}(a, l, h);;
    @ }
  @*/

Important points

▶ Why3/ACSL spec. lang. significantly diverged from JML
▶ Spec. language can be more powerful when RAC is not intended
▶ Yet, RAC may be useful to complement proofs
Design of E-ACSL

E-ACSL:

- Need for run-time checking in Frama-C
- *Executable* subset of ACSL
- Assertions turned into regular C code:
  - Mathematical integers handled using GMP
  - Built-in memory-related predicates (\valid, \initialized) handled using a specific memory management library
  - Axiomatic models not supported

ACSL and E-ACSL have slightly different semantics

Undefined expressions:

```
assert { 1/0 == 1/0 }
assert { *p == *p } // when p == NULL
```

- Valid in ACSL (logic of *total functions*)
- Raise errors in E-ACSL

Note: similar differences between JML RAC and ESC/Java
Ada contracts and SPARK 2014

Ada 2012:

- add contracts as part of regular Ada
- assertions are Boolean expressions
- Expression-functions can be used in assertions
- Bounded quantification now part of Ada expressions:
  
  \[
  \text{for all } I \text{ in } \langle \text{range} \rangle \Rightarrow P(I)
  \]

- \textit{Ada compiler generates corresponding run-time checks} for pre- and post-conditions
Static Verification in SPARK 2014

Important design choice

Semantics of annotations is fixed by the execution semantics

- VC are generated for well-definedness: 1/0, array index in bounds, etc.
- abstract models, unbounded integers:
  - not possible since it would forbids RAC
  - indeed possible via an SPARK-specific extension ("external axiomatization")
## Summary

<table>
<thead>
<tr>
<th></th>
<th>Why3</th>
<th>Frama-C ACSL</th>
<th>Frama-C E-ACSL</th>
<th>SPARK 2014</th>
</tr>
</thead>
<tbody>
<tr>
<td>Executable contracts</td>
<td>no</td>
<td>no</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>Only total functions in logic</td>
<td>yes</td>
<td>yes</td>
<td>no²</td>
<td>no²</td>
</tr>
<tr>
<td>Unbounded integers in logic</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes³</td>
</tr>
<tr>
<td>Unbounded quantification</td>
<td>yes</td>
<td>yes</td>
<td>no</td>
<td>no</td>
</tr>
<tr>
<td>Ghost code</td>
<td>yes</td>
<td>yes⁴</td>
<td>yes⁴</td>
<td>yes</td>
</tr>
</tbody>
</table>

1. run-time checks for well-definedness are generated
2. run-time checks and VCs for well-definedness are generated
3. possible through external axiomatization
4. restrictions: only executable C code, and non-interference with regular code is not checked

(See [Kosmatov et al., ISOLA’2016] for more details)
_ghost variable_: added to the regular, for the purpose of formal specification

_ghost code, subprograms_: extra code added to operate on ghost variables

**Ghost code**

Commonly used in most non-trivial examples
- keeping track of previous values of variables
- attach some abstract state (a kind of data refinement)
- etc.

Example: a sorting algorithm may return a ghost array of indices, giving the permutation of elements done by sorting.

```plaintext
procedure sort(a:array) returns (ghost p:array of integer)
    assigns a
    ensures \forall integer i; a[i]=\old(a)[p[i]]
```
Ghost code is possible in all of them

**Pros**

- Very useful in practice/for complex cases
- A kind of ’executable’ specification
- *Compatible with both static and run-time checking*

**Cons**

Tools should check non-interference between ghost code and regular code

- Why3, SPARK 2014 do it thanks to *strong non-aliasing policy*
- Frama-C doesn’t do it yet
Bonus: Lemma Functions

Proving theorems using ghost code!

\[
\text{ghost } f(x_1 : \tau_1, \ldots, x_n : \tau_n) \text{ returns } r : \tau
\]

\[
\text{requires } Pre
\]

\[
\text{ensures } Post
\]

if this function has \textit{no side-effect} and is \textit{proved terminating} then it is a constructive proof of

\[
\forall x_1, \ldots, x_n, \exists r, Pre \Rightarrow Post
\]

Examples:

- proving lemmas by induction (with automated provers only!)
- proving existential properties

Note: similar feature exists in other environments, e.g. Dafny
Outline

Why this joint Frama-C and SPARK day?
common history of Frama-C and SPARK

ACSL and SPARK 2014: how they differ?
static versus runtime checking
specification languages: design choices
advertisement: ghost code

Recent and Future Trends
bit-wise, floating-point computations
proof debugging, counterexamples
interactively discharging VCs
The ProofInUse project

Joint Lab between Inria and AdaCore

Main Goal
Spread the use of formal proof in SPARK users’ community

- Help for “debugging” when proof fails
  - Counterexamples
  - Simple interactive prover
- Enlarge language support
  - Bit-wise operators
  - Floating-point arithmetic
- Increase automation
  - Better exploit SMT solvers
Bit-Wise Operators

- New Why3 theory for bit-wise operations
- Use of *SMT-LIB bit-vector theory* (CVC4, Z3)
- Case study: *BitWalker*
  - Original C code by Siemens, ITEA 2 project OpenETCS
  - Rewritten by Jens Gerlach for Frama-C/WP
    - Formal specification in ACSL
    - proved with Alt-Ergo+Coq
  - Version in SPARK 2014
    - proved with Alt-Ergo+CVC4+Z3

See [Fumex et al., NFM’2016]
Counterexample Generation in SPARK

```ada
procedure Saturate (Val : in out Unsigned_16) is begin
    Val := Val and 16#FF#;
end Saturate;
```

Messages

Locations

Builder results (1 item)

saturation.adb (1 item)

6:7 medium: postcondition might fail (e.g. when
Counterexample Generation in SPARK

- Instrumentation of VC generation for tracing variables
- Query a model when SMT solver answers 'SAT'
- Reinterpret the model in the source code
- Display counterexample in the graphical interface

See [Hauzar et al., SEFM’2016]
Proof Debugging (Frama-C plug-in StaDy)

C code + annotations

Transformation A (non-compliance)

Transformation B (contract weakness)

C code

Dynamic Symbolic Execution

Report on annotations failures

- Non-compliance: code does not satisfy annotations
- Subcontract weakness: contracts of called functions, loop invariants, not powerful enough to prove the annotations correct

See [Petiot et al., TAP’2016]
Discharging VCs interactively

Goal
(hopefully simple) user interactions to assist automatic provers when proof fails

▶ On-going work for SPARK within ProofInUse joint lab
▶ Recently available in Frama-C/WP

See the talk by Loïc Correnson today!
Floating-Point Computations

Goals

- better handling Floating-Point in specifications and VC generation
- improve success rate of automated provers

- **SOPRANO** project
  - involves both Frama-C and SPARK developers
  - solvers Alt-Ergo FP and COLIBRI

- recent progress in SPARK
  - support for FP in SPARK 17.1, using
    - CodePeer interval analysis
    - FP support in prover Z3
  - on-going: use of Alt-Ergo FP and COLIBRI

See the talk by François Bobot today!
Conclusions

- Frama-C and SPARK share not only a common history but
  - A will to transfer academic research to the industry of critical software
  - Common challenges, approaches, technical solutions

OSIS Frama-C and SPARK day
Enjoy the talks, exchange ideas during breaks!