Using SMT solvers to prove SPARK VCs: Implications for the Rich Model Language

Paul Jackson

University of Edinburgh

IC0901 WG1&WG2 Meeting and FATPA Workshop
Belgrade
29th January 2010

Context

- SPARK
  - Ada subset: no heap, no recursion, no aliasing
  - Used in high-integrity applications (e.g. aerospace, security)
  - Promoted by ...
- (Altran) Praxis
  - Tool supplier: VC generator, automatic & interactive provers
  - Provider of consultancy services
    - *Correct by Construction* methodology
    - Extensive use of Z specifications
  - Example project: iFACTS — UK air traffic control

Research Opportunities

- Access to real-world case studies
- Development of better provers (both automatic and interactive)
  - Current proof focuses on exception freedom
  - Use of richer assertions deterred by
    - incompleteness of Praxis automatic prover
    - awkwardness of Praxis interactive prover
- Automatic invariant generation
- Proof explanation comprehensible to engineers
- Counter-example explanation

Current Work

- Have developed tool to translate SPARK VCs to
  - SMT-LIB language (logics with $\forall\exists, \mathbb{Z}, \mathbb{R}$, UFs)
  - Simplify language
  - API calls for Yices and CVC3
- Tool
  - gives SPARK users access to state-of-the-art SMT solvers
  - provides SMT solver developers with interesting benchmarks
- Tool in beta-test. GPL release available.
- Observations
  - Best SMT solvers faster and more complete than Praxis prover
  - SMT solver performance very sensitive to translation
  - Much variability in support for non-linear arithmetic
- Translations to HOL flavours and PVS on the way
- Developing non-linear arithmetic prover (Passmore)
  - Link-in likely to be indirect – via SMT solver

Translation Effort

- SPARK VC language has rich set of types: $\mathbb{Z}, \mathbb{R}, \{i..j\}$, ordered enumerations, records, arrays
- SMT-LIB types much simpler: $\mathbb{Z}, \mathbb{R}$, limited arrays
- Translation phases include
  1. Standardisation
  2. Enumerated type elimination
  3. Formula/term separation
  4. Type refinement
  5. Array & record elimination
  6. Boolean term elimination
  7. Arithmetic simplification
  8. Arithmetic elimination
  9. Type name & abstract type elimination

Translation example: refinement of array types

- Type refinement: elimination of subtypes

Bool, $\{0..9\}$	Int
Array $\{0..9\}$ of	Array Int of
$[\#$ fst : $\{0..9\}$, snd : Bool $\#]$	$[\#$ fst : Int, snd : Int $\#]$

- Type T can be modelled by type T^+ and unary predicate $\in_T$
- Consider $A \doteq \text{Array}\ I\ \text{of}\ E$ modelled by $A^+ \doteq \text{Array}\ I^+\ \text{of}\ E^+$
  - If $\in_A (a) \doteq \forall i : I^+.\ \in_I (i) \Rightarrow \in_E (a[i])$

    then a binary relation on A^+ for when two arrays are same is
    $a \equiv_A a' \doteq \forall i : I^+.\ \in_I (i) \Rightarrow a[i] =_E a'[i]$
  - However, if
    $\in_A (a) \doteq \forall i : I^+.\ (\in_I (i) \Rightarrow \in_E (a[i])) \wedge (\neg \in_I (i) \Rightarrow a[i] = k_E)$

    then OK to have $a \equiv_A a' \doteq a =_{A^+} a'$
  - In general, for many types T might need non-trivial $\equiv_T$ on T^+

Implications for a Rich Model Language

- Components of an RML might handle
  - Control structure
  - Structure of program state

  Will focus on data types for latter (e.g. after VC Gen has eliminated former)
- Tension
  - Prover developers want simple set of types
  - Users want richer set of types
- Should reconcile by building standard translation tools
  - Allows developers and users to focus on their primary work
  - Developers still free to work on support for richer types e.g. arrays, polymorphic types
- RML should be family of languages like *sub-logics* of SMT-LIB
- Part of RML could be extension of SMT-LIB

How rich could RML types be?

- No need to be too prescriptive
- Specification precision achieved with dependent types and subtypes
  - Subtypes for function domains avoid need for partial types
  - Both kinds feature in PVS, Nuprl, SAL, Yices 1, CVC3
- Use of subtypes has drawbacks
  - Lose decidability of typing
  - Need to prove *Type Correctness Conditions* in type checking
- Newer SMT solvers (Z3, Yices 2) moving away from these types
  - Any support must be external
- Use of set-theory rather than rich types sometimes advocated
  - Lose option of fast type checking of simply-typed skeletons
- Going too far?
  - Constructive logics & type theories (Why system . . .)
  - Isabelle/HOL or Haskell type classes