Automated Deductive Translation of Guardol Programs and Specifications into SMT-Provable Properties

Konrad Slind
Trusted Systems Group
Rockwell Collins Advanced Technology Center

May 8, 2013

Collaborators

- Rockwell Collins: David Hardin, Andrew Gacek
- U. Minnesota: Mike Whalen, Tuan-Hung Pham

Program Verification with SMT

People want to tap the power of SMT systems for program verification.

How to do this?

Assumption. The semantics of the programming language provides the basis for reasoning about individual programs.

Problem.
SMT systems don't understand semantics, they only understand the mathematical theories they support.

The Semantics Triad

Semantics-based program verification needs a translation between semantics and math.

- Axiomatic semantics (VC generation)
- Operational semantics (Decompilation into logic)
- Denotational semantics (Domain theory)

Other approaches

- Weakest-precondition semantics
- ACL2 (denotational semantics without tears)

Another Problem

- SMT doesn't understand recursion
- Ability to iterate leads to undecidability
- But lots of programs use loop structures or explicit recursion

There has been recent work on automating aspects of such induction proofs

Our Research

Our application area demands both a high level of assurance and a high level of automation

We have taken a two-pronged approach:

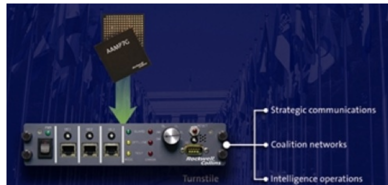
1. Express the operational semantics of our programming language in higher order logic and use decompilation to map programs to functional form
2. Raise the level of SMT proof to formulas incorporating a class of **catamorphisms** over algebraic datatypes

Application area: Guards

A **guard** mediates information sharing between security domains according to a specified policy.

Literally a box on a wire, in many cases.

Guard technology at Rockwell Collins



- Turnstile (2007)
- based on AAMP7 microprocessor
- rack size
- on UCDMO Baseline List

Guard technology at Rockwell Collins

- MicroTurnstile (2010)
- used to guard USB comms in soldier systems
- also AAMP7 based
- size of a stick of gum

Guard behaviors

Typical guard operations on a message stream:

- read field values in a message
- change fields in a message
- transform message by adding new fields
- drop fields from a message
- construct audit messages
- remove entire message from stream

Guard messages

Guards have traditionally been applied to fixed-size messages in low-level formats. These continue to be important.

HOWEVER

Guard functionality for for tree-structured data of arbitrary size (*e.g.*, email, XML) is increasingly needed.

Note that sanitizers for web browsers are also a kind of guard, applied to strings.

Guard properties

General properties:

- **live** : always terminates on every message
- **total** : never crashes because of a fault arising from processing a message

There are also guard-specific properties, *e.g.*, no dirty word gets past a dirty-word filter.

Guardol

We have designed and developed **Guardol**, a domain-specific language for guards.

Features:

- Automatic generation of implementations and formal analysis artifacts
- Integrated and highly automated formal analysis
- Ability to incorporate existing or mandated functionality
- Support for a wide variety of guard platforms

Guardol language summary

Guardol is designed to be a fairly simple language with cutting edge verification support.

- standard base types
- arrays, records
- standard imperative programming constructs
- ML style datatypes and pattern-matching
- declarations of external functionality
- a specification construct
- simple package system

Example: GMTI message format

```
type GMTI_Pkt =
 [ version : char[2],              -- 2 ASCII chars
   size: uint32,                   -- 4 bytes; includes 32-byte header
   nationality : char[2],          -- 2 ASCII chars
   classification : uint8,         -- 1 byte field
   classification_sys : char[2],   -- 2 ASCII chars
   security_code : uint16,         -- 2 bytes
   exercise_id : uint8,            -- 1 byte
   platform_id : char[10],         -- 10 ASCII chars
   mission_id : uint32,            -- 4 bytes
   job_id : uint32,                -- 4 bytes
   -- mission segment-------------------------
   mission_seg : uint8,            -- 1 byte
   mission_seg_size : uint32       -- 4 bytes
  -- wire_data : char [160]
 ] ;
```

GMTI Utility

```
function arrcpy(dest : inout char[dest_len],
                src : in uint8[src_len],
                src_offset : in uint,
                n : in uint) =
{
  if n=0 or n > dest_len or src_offset + n > src_len
    then skip;
    else
       for (i:uint = 0; i<n; i++)
       {
          dest[i] := chr(src[src_offset + i]);
       }
}
```

GMTI guard

```
function guard(pkt: in GMTI_Pkt) returns result: uint32 =
{
  result := 0;

  if (pkt.classification != 5)
   then result := result | (1 << 3);
   else skip;

  if not(pkt.classification_sys[0] = '#U' and
         pkt.classification_sys[1] = '#S')
    then result := result | (1 << 4);
    else skip;


  if ((pkt.security_code & 0xe010) != 0xe000)
    then result := result | (1 << 5);
    else skip;

  if (not((pkt.exercise_id = 128) or (pkt.exercise_id = 129)))
   then result := result | (1 << 6);
   else skip;

  ... <elided> ...
}
```

Example: Red-Black trees

Functional programming (nothing to do with guards)

```
type Color = { Red | Black } ;

type RBTree =
  { Leaf
  | Node : [color:Color,
            left:RBTree, elem:int, right:RBTree]
  } ;
```

Red-Black trees: Insertion

```
function ins(k:in int, T1:in RBTree) returns T2:RBTree =
{
  match T1
  { RBTree'Leaf =>
      T2 := RBTree'Node
              [color : 'Red,
               left:'Leaf, elem:k, right:'Leaf];
    RBTree'Node n =>
      if k < n.elem then
        T2 := balance(n.color, ins(k, n.left), n.elem, n.right);
      else if k = n.elem then
        T2 := T1;
      else
        T2 := balance(n.color, n.left, n.elem, ins(k, n.right));
  }
}
```

Red-Black trees: Insertion

```
function insert(Key: in int, T1 : in RBTree) returns T2 : RBTree =
{
  T2 := ins(Key, T1);
  match T2
  { RBTree'Leaf => skip;
    RBTree'Node n =>
      T2 := RBTree'Node
            [color : 'Black,
             left  : n.left,
             elem  : n.elem,
             right : n.right] ;
   }
}
```

No Two Reds

The children of a red node are both black.

```
function NoTwoReds(Tree: in RBTree) returns result: bool =
{
  match Tree
  {
    RBTree'Leaf => result := true;
    RBTree'Node n =>
    result := NoTwoReds(n.left) and NoTwoReds(n.right)
              and
              ((n.color = 'Black)
               or (colorOf(n.left) = 'Black and
                   colorOf(n.right) = 'Black));
  }
}
```

Red Property

Properties are stated with the **specification** construct and the **check** statement.

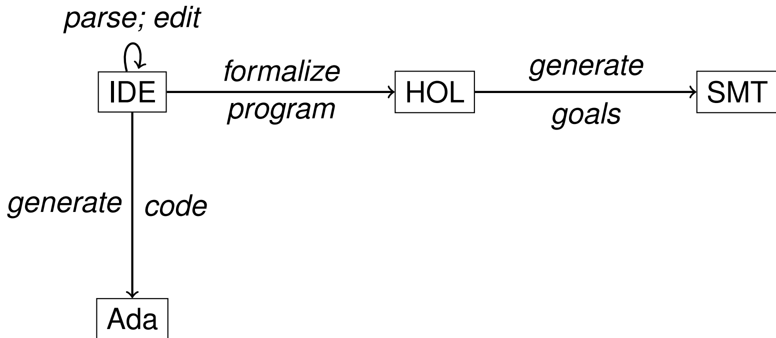
```
spec Red_Property = {
  var T1 : RBTree;
      T2 : RBTree;
      i : int;
  in
   if NoTwoReds(T1) then
    {
      T2 := insert(i,T1);
     check NoTwoReds(T2);
    }
   else skip;
 }
```

Note: no assertions, loop invariants, *etc.* in code.

What Guardol doesn't have

1. Infinite loops
   - A guard should always complete its task.
2. Pointers
   - Pointers complicate reasoning. Guardol provides automatic memory management for unbounded tree-shaped structures when generating code.
3. I/O
   - Guardol is aimed at just the guard, not its computational context, *i.e.*, how data gets to it, or how its output is dealt with.

The Guardol System



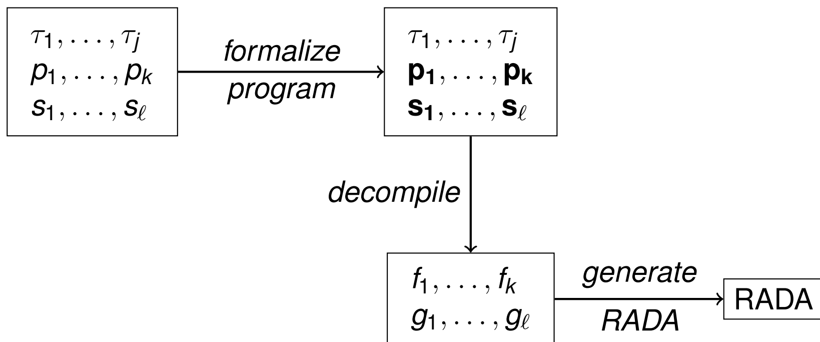
Verification

If the user chooses to verify Guardol programs, HOL4 and RADA become involved.

- **HOL4** is an implementation of higher order logic. It is well-suited to give semantics to programming languages.
- **RADA** is a SMT-based system for reasoning about catamorphisms

HOL is used as a **semantical conduit** to RADA

Verification path



Semantics notes

- HOL types directly represent Guardol types
- Extensibility of HOL type system models declaration of Guardol types
- Guardol statements are deeply embedded but expressions are not
- Based on SIMPL from Norbert Schirmer's PhD work

Guardol operational semantics

The operational semantics of Guardol is given as an inductively defined judgement saying how statements alter the program state. The formula:

$$\textbf{BIG}\ \Gamma\ \textit{prog}\ (\textbf{Normal}\ s_1)\ (\textbf{Normal}\ s_2)$$

says "evaluation of program *prog* beginning in state s_1 terminates and results in state s_2".

- A *big-step* semantics
- We also have small-step semantics and equivalence proof
- Γ is an environment binding procedure names to procedure bodies

Footprint functions

A **footprint** function models the effect of a piece of code on the program state.

We synthesize a footprint function for every procedure and specification declaration and make a HOL definition for it.

Recursive procedures result in definition of recursive functions.

(Point of failure when termination is not proved automatically!)

Decompilation into logic

A decompilation theorem

$$
\begin{array}{l}
\vdash \forall s_1\ s_2.\ \forall x_1 \ldots x_k. \\
\quad s_1.proc.v_1 = x_1 \wedge \ldots \wedge s_1.proc.v_k = x_k \wedge \\
\quad \textbf{BIG}\ \Gamma\ \textbf{code}\ (\textbf{Normal}\ s_1)\ (\textbf{Normal}\ s_2) \\
\quad \Rightarrow \\
\quad \texttt{let}\ (o_1, \ldots, o_n) = \textbf{fn}\ (x_1, \ldots, x_k) \\
\quad \texttt{in}\ s_2 = s_1\ \texttt{with}\ \{y_1 := o_1, \ldots, y_n := o_n\}
\end{array}
$$

relates evaluation of a program **code** with a footprint function **fn** which captures the behavior of the program.

Original idea in Myreen UCambridge PhD (2008).

Proving decompilation theorems

Decompilation theorems allow reasoning about execution to be replaced by reasoning about footprint functions.

- Automatically proved
- Essentially symbolic evaluation, using env. of decompilation theorems to summarize behavior of procedures
- Induction on recursion structure needed for recursive procedures.

Translating specifications

Recall the Red property for RBTree. It is explicitly stated in terms of evaluation.

```
spec Red_Property = {
   var T1 : RBTree;
       T2 : RBTree;
       i : int;
  in
    if NoTwoReds(T1) then
    {
      T2 := insert(i,T1);
      check NoTwoReds(T2);
    }
   else skip;
}
```

Transformation

Decompiling the code in the specification yields, by HOL proof, the following goal

$$\mathbf{NoTwoRedsFn}(v_2) \Rightarrow \mathbf{NoTwoRedsFn}(\mathbf{insertFn}(v_1, v_2))$$

The operational semantics is no longer present.

It has been melted away, but the connection with the original notion of program execution is preserved.

Deciding Guardol Specifications

We want to automate much or all of the reasoning about Guardol programs.

- In general impossible (Turing, Rice, *etc*)
- But, new decision procedures for functional programs over recursive datatypes have recently emerged
- We have implemented one of them, originally due to Suter and Kuncak (POPL 2010)
- The procedure dealts with catamorphisms mapping from algebraic datatypes to decidable theories

Catamorphisms

A catamorphism on lists is a simple pattern of recursion in which an operator

$$\textbf{op} : (\alpha \to \beta \to \beta) \to \alpha\ \textbf{list} \to \beta \to \beta$$

is used to crunch the list down into a single value.

$$\textbf{cata}\ (+)\ [x_1, \ldots, x_n]\ 0 = x_1 + \cdots + x_n + 0$$

Catamorphisms are definable for all algebraic datatypes.

Catamorphism Example

`NoTwoReds` is a catamorphism on `RBTree`.

```
function NoTwoReds(Tree: in RBTree) returns result: bool =
{
  match Tree
  {
    RBTree'Leaf => result := true;
    RBTree'Node n =>
    result := NoTwoReds(n.left) and NoTwoReds(n.right)
              and
              ((n.color = 'Black)
               or (colorOf(n.left) = 'Black and
                   colorOf(n.right) = 'Black));
  }
}
```

A decision procedure

Suppose a catamorphism $\mathbb{C}$ is *sufficiently surjective* and it reduces its arguments into a decidable theory.

Then formulas involving applications of $\mathbb{C}$ are also decidable.

Sufficient Surjectivity

Sufficient surjectivity is a semantic property.

Intuitively, a catamorphism is sufficiently surjective if the inverse relation of the catamorphism has sufficiently large cardinality when tree shapes are larger than some finite bound.

For example, the sum of a binary tree of numbers is s.s. because for any number there are an infinite number of trees summing to it.

Many common functions on trees are s.s.: *e.g.*, abstracting to a collection, size, height, min element, sortedness.

Further work

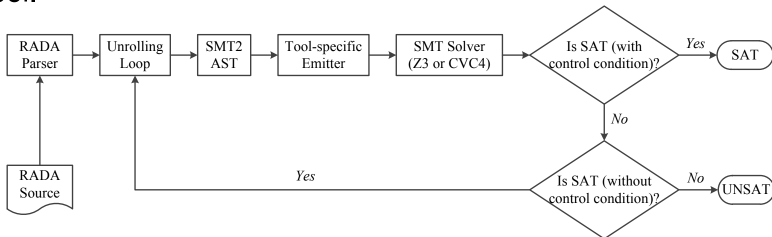
Whalen and Pham have further developed the theory behind the decision procedure

- Extended the procedure to handle mutual recursion
- Fixed completeness bugs
- Proposed *monotonic* catamorphisms (better than s.s.)
- Established explicit bounds
- Investigated combination of monotonic catamorphisms while preserving completeness

Paper in VSTTE 2013.

RADA architecture

The decision procedure has been implemented in the RADA tool.



RADA homepage

RADA can be obtained at

`http://crisys.cs.umn.edu/rada`

HOL meets RADA

Many properties of interest need to be proved by induction.

To orchestrate this, we have a **proof skeleton** in HOL4 that

- automatically picks a recursive function to induct on (as in Boyer-Moore)
- applies the corresponding induction scheme (proved in HOL)
- instantiates any ind. hyps. to remove quantifiers
- inserts previously proved specifications as lemmas

DEMO

Current and Future Work

- More accurate modelling of partial operations (array access exceptions, divide by zero, ...)
- Termination deferral (based on Greve and Slind, ACL2 Workshop 2013)
- Expand SK to a wider class of recursions, *e.g.*, real folds
- Translating SMT proofs
- Verifying binaries

Issue : dealing with failed proofs

Our proofs may fail for a variety of reasons

- Proof skeleton selects wrong induction scheme
- Resulting formulas not in decidable theory
- Co-domain restriction violated
- Property is not true

Finding a decent way to give feedback seems difficult

Summary

We have been developing a program generation and verification system based on a formal operational semantics.

Bridging the gap between the semantics and an SMT solver is achieved by

- decompilation into logic
- orchestrating the proof skeleton in higher order logic
- enhancing the SMT solver to deal with recursive functions

Bibliography

- *The Guardol Language and Verification System*, Hardin, Slind, Whalen, and Pham,TACAS 2012.
- *A DSL for Cross-Domain Security*, Hardin,Slind,Whalen and Pham,HILT 2012.
- *An Improved Unrolling-Based Decision Procedure for Algebraic Data Types*, Pham and Whalen,VSTTE 2013.

HILT 2103

Second High Integrity Language Technology Workshop. Pittsburgh, PA. November 10-14.

`http://www.sigada.org/conf/hilt2013/`

THE END